Contents

[01 Advanced C++ - const 2](#_Toc392180973)

[02 Advanced C++ - const and Functions 2](#_Toc392180974)

[03 Advanced C++ - Logic Constness and Bitwise Constness 3](#_Toc392180975)

[04 Advanced C++ - Compiler Generated Functions 4](#_Toc392180976)

[05 Advanced C++ - Disallow Functions 6](#_Toc392180977)

[06 Advanced C++ - Virtual Destructor and Smart Destructor 7](#_Toc392180978)

[07 Advanced C++ - Exceptions in Destructor 8](#_Toc392180979)

[08 Advanced C++ - Virtual Function in Constructor or Destructor 12](#_Toc392180980)

[09 Advanced C++ - Assignment to Self in Assignment Operator 13](#_Toc392180981)

[10 Advanced C++ - Resource Acquisition is Initialization 14](#_Toc392180982)

[11 Advanced C++ - Static Initialization Fiasco – Not there 15](#_Toc392180983)

[12 Advanced C++ - Struct Vs. Class 15](#_Toc392180984)

[13 Advanced C++ - Resource Managing Class 16](#_Toc392180985)

[14 Advanced C++ - Virtual Constructor - Clone() Function 18](#_Toc392180986)

[15 Advanced C++ - Define Implicit Type Conversion 20](#_Toc392180987)

[16 Advanced C++ - All Castings Considered - Part I 22](#_Toc392180988)

[17 Advanced C++ - All Castings Considered - Part II 24](#_Toc392180989)

[18 Advanced C++ - Inheritance - Public, Protected, and Private 28](#_Toc392180990)

[19 Advanced C++ - Maintain is-a Relation for Public Inheritance 29](#_Toc392180991)

[20 Advanced C++ - Understanding rvalue and lvalue 31](#_Toc392180992)

[21 Advanced C++ - Static Polymorphism – Not there 33](#_Toc392180993)

[22 Advanced C++ - Multiple Inheritance - Devil or Angel – Not there 33](#_Toc392180994)

[23 Advanced C++ - Duality Of Public Inheritance - Interface & Implementation 33](#_Toc392180995)

[24 Advanced C++ - Code Reuse - Inheritance vs Composition 34](#_Toc392180996)

[25 Advanced C++ - Namespace and Keyword using 36](#_Toc392180997)

[26 Advanced C++ - Koenig Lookup - Argument Dependent Lookup (ADL) 37](#_Toc392180998)

# 01 Advanced C++ - const

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

const:

- A "compile time" constraint that an object cannot be modified.

Benefits

a) Guards against inadvertent write to the variable.

b) Self documenting - Telling the reader of the code that the variable

is not modified.

c) Enables compiler to do more optimization, making code tighter.

d) const means the variable can be put in ROM.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(void)

{

const int i = 9;

//i = 6; // expression must be a modifiable lvalue

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

If const is on the left of \*, data is const

If const is on the right of \*, pointer is const

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const int aConstInt = 9;

// If we want to change the value of aConstInt, we can use const\_cast

const\_cast<int&>(aConstInt) = 6;

int j = 9; // j is not const.

// static\_cast will make the j const

// static\_cast<const int&>(j) = 7;

return 0;

}

# 02 Advanced C++ - const and Functions

class Dog

{

private:

int \_age;

string \_name;

public:

Dog() { \_age = 3; \_name = "Bruce"; }

// const parameters

/\* Function which take reference parameter can be overloaded with the

function which takes const reference parameter.

\*/

void setAge(const int& a) { \_age = a; } // So that a doesn't get modified.

// void setAge(const int a) { \_age = a; } // No use of const here, cos value is passed by value

// const return value

/\* Returns const string reference. No copy is made and the

caller can't change the value of \_name \*/

const string& getName() { return \_name; }

/\* Below, const is completely useless. Because \_name is returned

by value. \*/

// const string getName() { return \_name; }

// const function

/\*

- Below function promises not to modify any member variables.

- A const function can only call another const function.

- const can be used to overload a function

\*/

// Below function will be called by const Dog

void printDogName() const { cout << \_name << ":: const function" << endl; }

// Below function will be called by Non-const Dog

void printDogName() { cout << \_name << ":: Non-const function" << endl; }

};

int main(void)

{

Dog d;

int i = 9;

d.setAge(i);

const string& name = d.getName();

cout << i << endl;

cout << name << endl;

d.printDogName();

const Dog d1;

d1.printDogName();

return 0;

}

# 03 Advanced C++ - Logic Constness and Bitwise Constness

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Bitwise const-ness/Physical const-ness is what provided by C++ compiler.

- This checks whether any member variable is getting modified or not.

Logical const-ness is what is our model.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class BigArray

{

private:

vector<int> aVector; // Huge Vector

int accessCounter; // How many times the vector has been accessed.

int\* v2; // Another int array

public:

int getItem(int index) const

{

// Using mutable, we can modify accessCounter.

//accessCounter++;

// Using const\_cast we can modify accessCounter.

const\_cast<BigArray\*>(this)->accessCounter++;

return aVector[index];

}

void setV2Item(int index, int x) const

{

\*(v2 + index) = x;

}

};

int main(void)

{

BigArray b;

return 0;

}

# 04 Advanced C++ - Compiler Generated Functions

#pragma region CompilerGeneratedFunctions

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

- Compiler silently generates 4 function if they are not explicitly

declared by us:

1. Copy Constructor.

2. Copy Assignment Operator.

3. Destructor.

4. Default Constructor (Only if there is no constructor declared).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class Dog { /\*Empty Definition\*/ };

//Is Equal to below declared class

class Dog

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

- What if one of the member is const reference? Then copy is not possible.

Then compiler doesn't generate the operator= function.

- What if the base class doesn't have the default constructor. Then the compiler

will not generate the default constructor for the class.

- What if the base class destructor is private? Then compiler will not generate

the default destructor.

- Compiler generated functions are public and inline.

- They are generate only if they are needed.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public:

Dog(const Dog& rhs) { /\* Member by member initialization \*/ }

Dog& operator=(const Dog& rhs) {/\* Member by member copy \*/}

Dog(){ } // 1. Call base class's default constructor.

// 2. Call data member's default constructor.

~Dog() { } // 1. Call base class's destructor.

// 2. Call data member's destructor.

};

#pragma endregion CompilerGeneratedFunctions

#pragma region Example

class dog

{

public:

string \_name;

// Default constructor.

dog(string name = "Bob")

{

\_name = name;

cout << name << " is born." << endl;

}

~dog()

{

cout << \_name << " is destroyed";

}

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Copy Constructor - Copy constructor is not used in main, so It is

not generated by compiler.

2. Copy Assignment Operator - We have used copy assignment and we haven't

declared any copy assignment operator. Compiler will generate one for us.

3. Destructor - Dog already has a destructor defined. So compiler will not

generate any destructor for us.

\*\*\*A default constructor is a CONSTRUCTOR which can work without any parameter.\*\*\*

4. Default Constructor - We are constructing an object with default

constructor and we have defined a default constructor in our class. So

it will not be generated for us.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(void)

{

dog dog1{ "Henry" };

dog dog2;

dog2 = dog1;

return 0;

}

#pragma endregion Example

#pragma region AnotherExample

class collar

{

public:

collar(string color) { cout << " collar is born!!!\n"; }

};

class dog

{

collar \_collar;

};

int main(void)

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Below code wont compile:

- We are creating a dog1, I am calling the default constructor of Dog

and my dog doesn't have default constructor, compiler will try to create

a default constructor for me.

- We know that, compiler generated default constructor tries to call

the DATA members default constructor (In our case, \_collar default

constructor). But collar doesn't have a default constructor either.

Compiler tries to generate default constructor for collar first. Since

collar already have a constructor which takes string parameter, the

compiler will not be able to generate a default constructor for collar.

As a result, compiler will not be able to generate a default constructor

for dog either. So will get an error message...

'dog' : no appropriate default constructor available

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

dog dog1;

}

#pragma endregion AnotherExample

# 05 Advanced C++ - Disallow Functions

class openFile

{

public:

// If we define a constructor which takes a parameter, compiler will not

// generate any default constructor for me.

openFile(string fileName){ cout << "Open a file " << fileName << endl; }

// In C++11, we can disallow copy constructor by using "delete" as shown

// below or make the copy constructor private and don't define the function.

openFile(openFile& rhs) = delete;

// Disallow the copy assignment operator.

openFile& operator=(const openFile& rhs) = delete;

// Say this openFile class has inherited the function "writeLine()" and

// we don't want the write line to be used by openFile class, then we can

// declare the write line function as private and don't provide definition

// as shown below...

private:

void writeLine(string str);

// Class with a destructor in the private can only be instantiated on heap

// and can't be stored on stack. Let heavy weight classes to have private

// destructor (in embedded programming)

//~openFile() { cout << "openFile Destructor!!!" << endl; }

};

int main(void)

{

// We need name of the file to open a file, so don't allow

// compiler to generate a default constructor for me.

//openFile f; // Fails!!!

openFile f("aTextFile.txt");

// Below code will call the compiler generated copy constructor.

// f2 also starts writing to a file. So disallow the copy constructor.

//openFile f2(f); // Fails!!!

openFile f2("bTextFile.txt");

// Below code will call the compiler generated copy assignment operator.

// because of this f also starts writing to a file. So disallow

// the copy assignment operator.

//f = f2; // Fails!!!

return 0;

}

# 06 Advanced C++ - Virtual Destructor and Smart Destructor

#pragma region VirtualDestructor

class Dog

{

public:

// virtual ~Dog() { cout << "Dog destroyed!!!" << endl; }

~Dog() { cout << "Dog destroyed!!!" << endl; }

};

class YellowDog : public Dog

{

public:

~YellowDog() { cout << "Yellow dog destroyed!!!" << endl; }

};

class DogFactory

{

public:

static Dog\* createYellowDog() { return (new YellowDog()); }

// Create other dogs

};

int main(void)

{

Dog\* ptrDog = DogFactory::createYellowDog();

// 1. When ptrDog gets deleted, only Dog's destructor gets called this

// is bad, because we have created YellowDog and only a part of YellowDog

// gets destroyed. This is why we need virtual destructor for base class

delete ptrDog;

return 0;

}

#pragma endregion VirtualDestructor

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

All the classes in STL have no virtual destructor, so we should be

careful when inheriting from them. We better use shared\_ptr for that.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma region Using shared\_ptr

class Dog

{

public:

~Dog() { cout << "Dog destroyed!!!" << endl; }

};

class YellowDog : public Dog

{

public:

~YellowDog() { cout << "Yellow dog destroyed!!!" << endl; }

};

class DogFactory

{

public:

//static Dog\* createYellowDog() { return (new YellowDog()); }

static shared\_ptr<Dog> createYellowDog()

{

return shared\_ptr<YellowDog>(new YellowDog());

}

// Create other dogs

};

int main(void)

{

// If we don't want to have virtual destructor, then we can use

// shared\_ptr

shared\_ptr<Dog> ptrDog = DogFactory::createYellowDog();

// shared\_ptr is responsible to delete the pointer.

//delete ptrDog;

return 0;

}

#pragma endregion Using shared\_ptr

# 07 Advanced C++ - Exceptions in Destructor

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Preventing exceptions from leaving destructor

- Exceptions thrown by destructor should be contained in the

destructor and should not be thrown out of destructor.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma region Analysis

class Dog

{

public:

string \_name;

Dog(string name) : \_name(name) { cout << name << " is born!!!" << endl; }

~Dog() { cout << \_name << " is destroyed!!!" << endl; }

void bark(void) { cout << \_name << " is barking!!!" << endl; }

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

OUTPUT

Henry is born!!!

Bob is born!!!

Bob is destroyed!!!

Henry is destroyed!!!

20 is caught

- Main point here is, before the exception 20 is caught by the catch

block, stack unwinding happens and the objects gets destroyed.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(void)

{

try

{

Dog aDog("Henry");

Dog bDog("Bob");

throw 20;

aDog.bark();

bDog.bark();

}

catch (int e)

{

cout << e << " is caught" << endl;

}

return 0;

}

#pragma endregion Analysis

#pragma region Problem

class Dog

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

PROGRAM CRASHES!!!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public:

string \_name;

Dog(string name) : \_name(name) { cout << name << " is born!!!" << endl; }

void bark(void) { cout << \_name << " is barking!!!" << endl; }

// An exception is added in the destructor.

~Dog()

{

cout << \_name << " is destroyed!!!" << endl;

throw 20;

}

};

int main(void)

{

// An exception is added in the destructor.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Why the program crashes?

When the execution reaches the end of try block, stack unwinding happens and

 all the local variables needs to be destroyed. So, dDog("Bob")  will be

 destroyed first. When the Bob destructor is executed, it throws an

 exception. So before the exception is caught by the catch block, aDog("Henry")

 also needs to be destroyed. So aDog destructor also executes and also throws

 an exception.

- As a result, there are 2 exceptions pending at the same time one from Bob

and one from Henry. C++ doesn't handle 2 exceptions at the same time so

the program crashes.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

try

{

Dog aDog("Henry");

Dog bDog("Bob");

aDog.bark();

bDog.bark();

}

catch (int e)

{

cout << e << " is caught" << endl;

}

return 0;

}

#pragma endregion Problem

#pragma region Solution 1

class Dog

{

public:

string \_name;

Dog(string name) : \_name(name) { cout << name << " is born!!!" << endl; }

void bark(void) { cout << \_name << " is barking!!!" << endl; }

~Dog()

{

try

{

cout << \_name << " is destroyed!!!" << endl;

throw 20;

}

// Enclose all the exception prone code here

catch (/\*MYEXCEPTION e\*/int e)

{

cout << "Exception is thrown in the destructor!!! " << e << endl;

}

catch (...)

{

cout << "Exception is thrown in the destructor!!!";

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

- Downside of this solution is that the exception is

swallowed by the Dog's destructor. The Dog's client

will not get the exception and the Dog's client will

not know what has happened!!!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

}

};

int main(void)

{

try

{

Dog aDog("Henry");

Dog bDog("Bob");

aDog.bark();

bDog.bark();

}

catch (int e)

{

cout << e << " is caught" << endl;

}

return 0;

}

#pragma endregion Solution 1

#pragma region Sloution 2 MoveExceptionProneCode

class Dog

{

public:

string \_name;

Dog(string name) : \_name(name) { cout << name << " is born!!!" << endl; }

void bark(void) { cout << \_name << " is barking!!!" << endl; }

~Dog()

{

cout << \_name << " is destroyed!!!" << endl;

}

void prepareToDestruct()

{

cout << "Throwing exception!!!" << endl;

// Moving the exception prone code to other method

throw 20;

}

};

int main(void)

{

try

{

Dog aDog("Henry");

Dog bDog("Bob");

aDog.bark();

bDog.bark();

aDog.prepareToDestruct(); // Another API needs to be called by Dog's client

bDog.prepareToDestruct(); // Another API needs to be called by Dog's client

}

catch (int e)

{

cout << e << " is caught" << endl;

}

return 0;

}

#pragma endregion Sloution 2 MoveExceptionProneCode

# 08 Advanced C++ - Virtual Function in Constructor or Destructor

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Pitfall in calling Virtual Function in Constructor and Destructor

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class Dog

{

public:

Dog()

{

cout << "Dog born" << endl;

// Invoking bark function

// Behaves like non virtual function since the Yellow dog is not yet born

bark();

}

virtual void bark(void){ cout << "I am just a dog!!!" << endl; }

void seeCat(void){ bark(); }

// Invoking bark function

// Behaves like non virtual function since the Yellow dog is already destroyed

~Dog() { bark(); }

};

class YellowDog : public Dog

{

public:

YellowDog(){ cout << "Yellow dog born" << endl; }

virtual void bark(){ cout << "I am an yellow dog" << endl; }

};

int main(void)

{

YellowDog yd;

yd.seeCat();

return 0;

}

# 09 Advanced C++ - Assignment to Self in Assignment Operator

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Assignment of self in assignment operator

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma region Solution 1

class collar

{

public:

string \_collar;

};

class Dog

{

collar\* ptrCollar;

public:

Dog& operator=(const Dog& rhs)

{

if (this == &rhs)

return \*this;

//delete ptrCollar;

// What happens if the copy constructor throws an exception?

// The ptrCollar has already been deleted. In order to avoid this

// delete ptrCollar after successful construction.

collar\* pOriginalCollar = ptrCollar;

ptrCollar = new collar(\*rhs.ptrCollar);

delete pOriginalCollar;

return \*this;

}

};

#pragma endregion Solution 1

#pragma region Solution 2

class collar

{

public:

string \_collar;

};

class Dog

{

collar\* ptrCollar;

public:

// Delegates the assignment operator to collar class.

Dog& operator=(const Dog& rhs)

{ // Member by member copying of collar

\*ptrCollar = \*rhs.ptrCollar;// OR

} // call collar's operator=

};

#pragma endregion Solution 2

# 10 Advanced C++ - Resource Acquisition is Initialization

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Resource Acquisition is Initialization

mutex myMutex = MUTEX\_INITIALIZER;

void aFunction(void)

{

myMutex.lock();

// Do a bunch of things

myMutex.unlock(); // Will this line always be executed????

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma region Example 1

class Lock

{

private:

mutex\* myMutex;

public:

explicit Lock(mutex\* ptr)

{

// Lock the mutex here

}

~Lock()

{

// Unlock the mutex here

}

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

- The only code that can be guaranteed to be executed after exception

is thrown are the destructor's of the objects residing on stack.

- Resource management therefore needs to be tied to the lifespan of

suitable objects in order to gain automatic deallocation and reclamation.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma endregion Example 1

#pragma region Example 2

class Dog

{

public:

int age;

string name;

};

void aFunction()

{

// A shared pointer is reference counted smart pointer.

// shared pointer is destroyed when the reference count reaches zero.

shared\_ptr<Dog> ptrDog(new Dog);

}

#pragma endregion Example 2

# 11 Advanced C++ - Static Initialization Fiasco – Not there

# 12 Advanced C++ - Struct Vs. Class

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Struct Vs. Class

- The members of structure are public by default but for class,

its private. This is the only difference as far as language is concerned.

Semantic Difference:

- Structure is used for small passive object that carry public data

and have no or few basic member functions. In other words structure is

used for data container.

- Class is used for bigger active objects that carry private data,

interfaced through public member functions. In other words class is

used for more complex data structure.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

struct StructPerson

{

// Public by default

string name;

unsigned age;

StructPerson(unsigned personAge = 5) : age(personAge) {}

};

class ClassPerson

{

// Private by default

string name\_; // Trailing underscore notation is used for naming class data members.

unsigned age\_;

public:

// Too many getter and setter means there is some design problem.

unsigned getAge(void) const { return age\_; } // Getter or accessors

void setAge(unsigned age) { age\_ = age; } // Setter or mutator

};

int main(void)

{

StructPerson aPerson;

cout << "Person age :" << aPerson.age << endl;

ClassPerson bPerson;

bPerson.setAge(4);

cout << "Person age :" << bPerson.getAge() << endl;

return 0;

}

# 13 Advanced C++ - Resource Managing Class

#pragma region TheProblem

// Person owns the string through its pointer

class Person

{

public:

Person(string name) { ptrname\_ = new string(name); }

~Person() { delete ptrname\_; }

// CRASHES!!!

void printName(void) { cout << \*ptrname\_; }

unsigned getAge(void) const { return age\_; }

void setAge(unsigned age) { age\_ = age; }

private:

string\* ptrname\_;

unsigned age\_;

};

int main(void)

{

vector<Person> persons;

persons.push\_back(Person("Nishith"));

// Older C++ version

// 1. "Nishith" is constructed.

// 2. A copy of "Nishith" is saved in the vector persons (shallow copy)

// 3. "Nishith" is destroyed because it is rvalue.

persons.back().printName();

cout << "Goodbye!!!" << endl;

return 0;

}

#pragma endregion TheProblem

// Solution 1: Define copy constructor and copy assignment operator for deep copying

#pragma region Solution1

// Person owns the string through its pointer

class Person

{

public:

Person(string name) { ptrname\_ = new string(name); }

~Person() { delete ptrname\_; }

void printName(void) { cout << \*ptrname\_; }

unsigned getAge(void) const { return age\_; }

void setAge(unsigned age) { age\_ = age; }

Person(const Person&rhs)

{

ptrname\_ = new string(\*(rhs.ptrname\_));

}

Person& operator=(const Person& rhs)

{

}

private:

string\* ptrname\_;

unsigned age\_;

};

int main(void)

{

vector<Person> persons;

persons.push\_back(Person("Nishith"));

// Older C++ version

// 1. "Nishith" is constructed.

// 2. A copy of "Nishith" is saved in the vector persons (shallow copy)

// 3. "Nishith" is destroyed because it is rvalue.

persons.back().printName();

cout << "Goodbye!!!" << endl;

return 0;

}

#pragma endregion Solution1

// Solution 2: Delete copy constructor and copy assignment operator, define clone function

#pragma region Solution 2

// Person owns the string through its pointer

class Person

{

public:

Person(string name) { ptrname\_ = new string(name); }

~Person() { delete ptrname\_; }

// CRASHES!!!

void printName(void) { cout << \*ptrname\_; }

unsigned getAge(void) const { return age\_; }

void setAge(unsigned age) { age\_ = age; }

Person(const Person&rhs) = delete;

Person& operator=(const Person& rhs) = delete;

Person\* clone()

{

return (new Person(\*(ptrname\_)));

}

private:

string\* ptrname\_;

unsigned age\_;

};

int main(void)

{

vector<Person> persons;

persons.push\_back(Person("Nishith"));

// Older C++ version

// 1. "Nishith" is constructed.

// 2. A copy of "Nishith" is saved in the vector persons (shallow copy)

// 3. "Nishith" is destroyed because it is rvalue.

persons.back().printName();

cout << "Goodbye!!!" << endl;

return 0;

}

#pragma endregion Solution 2

# 14 Advanced C++ - Virtual Constructor - Clone() Function

#include <iostream>

// Co-variant return type

using namespace std;

#pragma region TheProblem

class Dog

{

public:

Dog(){ cout << "Inside Dog constructor" << endl; }

};

class Yellowdog : public Dog

{

public:

Yellowdog(){ cout << "Inside Yellow dog constructor" << endl; }

};

// What afunction  gets is an yellow dog.

void aFunction(Dog\* d)

{

// Because of copy construction, c will become a dog.

Dog\* c = new Dog(\*d);

//...

// Play with dog c

}

int main(void)

{

Yellowdog d;

aFunction(&d);

return 0;

}

#pragma endregion TheProblem

#pragma region Solution

class Dog

{

public:

Dog(){ cout << "Inside Dog constructor" << endl; }

// Co-variant return type

virtual Dog\* clone(){ return (new Dog(\*this)); }

};

class Yellowdog : public Dog

{

public:

Yellowdog(){ cout << "Inside Yellow dog constructor" << endl; }

// Co-variant return type

virtual Yellowdog\* clone(){ return (new Yellowdog(\*this)); }

};

// What afunction  gets is an yellow dog.

void aFunction(Dog\* d)

{

// Because of copy construction, c will become a dog.

//Dog\* c = new Dog(\*d);

Dog\* c = d->clone();

//...

// Play with dog c

}

int main(void)

{

Yellowdog d;

aFunction(&d);

return 0;

}

#pragma endregion Solution

# 15 Advanced C++ - Define Implicit Type Conversion

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

User Defined Implicit Type Conversion

Categories of type conversion: Implicit Explicit

Standard Type Conversion A B

User Defined Type Conversion C D

(Casting)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma region Category A: Implicit Standard Type Conversion

class Dog

{

public:

Dog(){ cout << "Inside Dog constructor" << endl; }

};

class Yellowdog : public Dog

{

public:

Yellowdog(){ cout << "Inside Yellow dog constructor" << endl; }

};

int main(void)

{

{

char c = 'A';

int i = c; // Integral promotion

char\* ptrC = 0; // int->Null pointer

void aFunction(int i);

aFunction(c);

Dog\* pd = new Yellowdog();

}

return 0;

}

#pragma endregion Category A: Implicit Standard Type Conversion

#pragma region Category C: Implicit User Define Type Conversion

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Defined inside class

Method 1: Use constructor that can accept a single parameter and convert

other types of objects into your class.

Method 2: Use the type conversion function and convert an object of your

class into other types.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class Dog

{

public:

// Constructor which takes a single parameter is not just constructor

// its a constructor + type converter. In the below case, it converts string to Dog

Dog(string name){ name\_ = name; }

string getName(){ return name\_; }

// Type Conversion Function

operator string() const { return name\_; }

private:

string name\_;

};

class Yellowdog : public Dog

{

public:

Yellowdog(): Dog("Bruce") { cout << "Inside Yellow dog constructor" << endl; }

};

int main(void)

{

string dogName = "Bob";

Dog dog1 = dogName; // Construction a Dog with string

// If we don't want to use getName(), then go for Method 2

cout << "My dog name is :" << dog1.getName() << endl;

cout << "My dog name is :" << dog1 << endl;

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

General guideline:

1. Avoid defining seemingly unexpected conversion.

2. Avoid defining two-way implicit conversion.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma endregion Category C: Implicit User Define Type Conversion

# 16 Advanced C++ - All Castings Considered - Part I

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

CASTING

Type Conversion:

1. Implicit Type Conversion

2. Explicit Type Conversion - also called CASTING

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class Dog

{

public:

Dog(string name){ cout << "Inside Dog constructor" << endl; }

operator string() { return "Hello"; }

virtual ~Dog(){}

};

class Yellowdog : public Dog

{

public:

Yellowdog(): Dog("Hello") { cout << "Inside Yellow dog constructor" << endl; }

};

int main(void)

{

#pragma region StaticCast

{

// Static Cast will work as long as type converter's are defined for those types.

int i = 9;

float f = static\_cast<float> (i);

Dog d1 = static\_cast<Dog> (string("BOB")); // Type converter is defined here

Dog\* ptrDog = static\_cast<Dog\*> (new Yellowdog());

}

#pragma endregion StaticCast

#pragma region DynamicCast

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Dynamic cast can work only on pointers or reference, it can't work

on objects.

2. It converts one type to a related type (Typically used in Down cast)

3. Does run-time type check. If not compatible, then returns nullptr.

4. Ti requires the two types to be polymorphic.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

{

Dog\* ptrDog = new Yellowdog();

Yellowdog\* ptrYelloDog = dynamic\_cast<Yellowdog\*>(ptrDog);

}

#pragma endregion DynamicCast

#pragma region ConstCast

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Works only on pointers or references can't work on objects.

2. Only works on same type.

3. To cast away constness of the object being pointed to.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

{

const char\* str = "Hello World";

char\* modifiable = const\_cast<char\*>(str);

}

#pragma endregion ConstCast

#pragma region ReinterpretCast

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Works only on pointers or references can't work on objects.

2. Can work on any type!!!!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

{

long p = 51110900;

Dog\* dd = reinterpret\_cast<Dog\*>(p);

}

#pragma endregion ReinterpretCast

#pragma region C-StyleCasting

// Mixture of static\_cast, const\_cast and reinterpret\_cast

{

short a = 2000;

int i = (int)a; // C-Like cast notation

int j = int(a); // Functional notation

}

#pragma endregion C-StyleCasting

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Generally C++ style of casts are preferred over C-Style, because:

1. Easier to identify in the code.

2. Less usage error.

a. Narrowly specified purpose for each cast.

b. Run-time type check capability.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

return 0;

}

# 17 Advanced C++ - All Castings Considered - Part II

#pragma region TheProblem

class Dog

{

public:

Dog(){ /\*cout << "Inside Dog constructor" << endl;\*/ }

virtual ~Dog(){}

};

class Yellowdog : public Dog

{

int age\_;

public:

Yellowdog(){ /\*cout << "Inside Yellow dog constructor" << endl;\*/ }

void bark(){ cout << "Woof...Woof..." << endl; }

};

int main(void)

{

Dog\* ptrDog = new Dog;

// We are casting Dog to Yellow Dog...

Yellowdog\* ptrYellowDog = dynamic\_cast<Yellowdog\*> (ptrDog);

// YellowDog will print nullptr here, and causes undefined behavior.

// Since we are not accessing any member variable inside bark, this function

// can be treated as static function and still this call will print Woof...Woof

ptrYellowDog->bark();

// YellowDog will print nullptr here

cout << "ptrYellowDog = " << ptrYellowDog << endl;

cout << "ptrDog = " << ptrDog << endl;

return 0;

}

#pragma endregion TheProblem

#pragma region AnotherProblem

class Dog

{

public:

Dog(){ /\*cout << "Inside Dog constructor" << endl;\*/ }

virtual ~Dog(){}

};

class Yellowdog : public Dog

{

int age\_;

public:

Yellowdog(){ age\_ = 0;/\*cout << "Inside Yellow dog constructor" << endl;\*/ }

void bark(){ cout << "Woof...Woof..." << age\_ << endl; }

};

int main(void)

{

Dog\* ptrDog = new Dog;

// We are casting Dog to Yellow Dog... All yellow dogs are dogs, but all

// dogs are not yellow dogs.

Yellowdog\* ptrYellowDog = dynamic\_cast<Yellowdog\*> (ptrDog);

// YellowDog will be nullptr here, and when we try to access the member variable of

// Yellow Dog, the program CRASHES!!!!!!!!!!.

ptrYellowDog->bark();

// YellowDog will print nullptr here

cout << "ptrYellowDog = " << ptrYellowDog << endl;

cout << "ptrDog = " << ptrDog << endl;

return 0;

}

#pragma endregion AnotherProblem

#pragma region Using StaticCast

class Dog

{

public:

Dog(){ /\*cout << "Inside Dog constructor" << endl;\*/ }

virtual ~Dog(){}

};

class Yellowdog : public Dog

{

int age\_;

public:

Yellowdog(){ age\_ = 0;/\*cout << "Inside Yellow dog constructor" << endl;\*/ }

void bark(){ cout << "Woof...Woof..." << age\_ << endl; }

};

int main(void)

{

Dog\* ptrDog = new Dog;

// static\_cast will always succeed

Yellowdog\* ptrYellowDog = static\_cast<Yellowdog\*> (ptrDog);

ptrYellowDog->bark();

cout << "ptrYellowDog = " << ptrYellowDog << endl;

cout << "ptrDog = " << ptrDog << endl;

return 0;

}

#pragma endregion Using StaticCast

#pragma region Solution 1

class Dog

{

public:

Dog(){ /\*cout << "Inside Dog constructor" << endl;\*/ }

virtual ~Dog(){}

};

class Yellowdog : public Dog

{

int age\_;

public:

Yellowdog(){ age\_ = 0;/\*cout << "Inside Yellow dog constructor" << endl;\*/ }

void bark(){ cout << "Woof...Woof..." << age\_ << endl; }

};

int main(void)

{

Dog\* ptrDog = new Dog;

Yellowdog\* ptrYellowDog = dynamic\_cast<Yellowdog\*> (ptrDog);

if (ptrYellowDog)

ptrYellowDog->bark();

cout << "ptrYellowDog = " << ptrYellowDog << endl;

cout << "ptrDog = " << ptrDog << endl;

return 0;

}

#pragma endregion Solution 1

#pragma region Better Solution

class Dog

{

public:

Dog(){ /\*cout << "Inside Dog constructor" << endl;\*/ }

virtual ~Dog(){}

// Better solution is to add the virtual bark function in the base class

virtual void bark(){}

};

class Yellowdog : public Dog

{

int age\_;

public:

Yellowdog(){ age\_ = 0;/\*cout << "Inside Yellow dog constructor" << endl;\*/ }

void bark(){ cout << "Woof...Woof..." << age\_ << endl; }

};

int main(void)

{

Dog\* ptrDog = new Dog;

Yellowdog\* ptrYellowDog;

//Yellowdog\* ptrYellowDog = dynamic\_cast<Yellowdog\*> (ptrDog);

//if (ptrYellowDog)

ptrYellowDog->bark();

cout << "ptrYellowDog = " << ptrYellowDog << endl;

cout << "ptrDog = " << ptrDog << endl;

return 0;

}

#pragma endregion Better Solution

class Dog

{

public:

string name\_;

Dog() : name\_("Bruce") { /\*cout << "Inside Dog constructor" << endl;\*/ }

// Literally a function can't be const. Then who is const here in the

// below code? Answer: \*this is const

void bark() const

{

cout << "My name is " << name\_ << endl;

const\_cast<Dog\*>(this)->name\_ = "Henry";

}

};

int main(void)

{

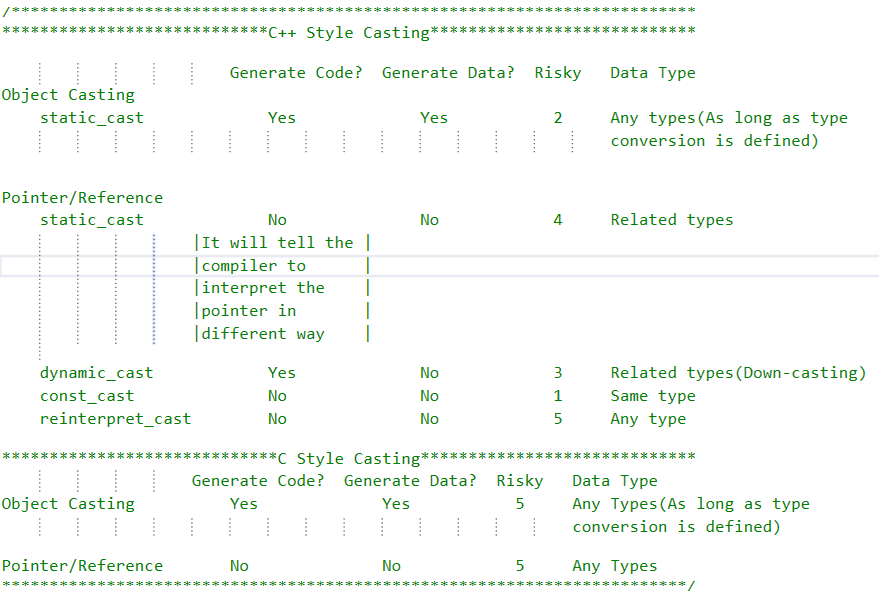
Dog aDog;

aDog.bark();

cout << "Now my name is: " << aDog.name\_ << endl;

return 0;

}



# 18 Advanced C++ - Inheritance - Public, Protected, and Private

#include <iostream>

#include <string>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Inheritance - Public, Protected, and Private

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class Base

{

public:

void f\_Public(){ cout << "f\_Public() is called " << endl; }

protected:

void f\_Protected(){ cout << "f\_Protected is called " << endl; }

private:

void f\_Private(){ cout << "f\_Private is called " << endl; }

};

class Derived\_Public : public Base

{

public:

void aFunction()

{

f\_Public(); // OK, f\_Public is Derived\_Public's public function

f\_Protected(); // OK, f\_Protected is Derived\_Public's public function

//f\_Private(); // ERROR, f\_Private is Base's private function

}

};

class Derived\_Protected : protected Base

{

public:

void aFunction()

{

f\_Public(); // OK, f\_Public is Derived\_Protected's protected function

f\_Protected(); // OK, f\_Protected is Derived\_Protected's protected function

//f\_Private(); // ERROR, f\_Private is Base's private function

}

};

class Derived\_Private : private Base

{

public:

void aFunction()

{

f\_Public(); // OK, f\_Public is Derived\_Private's private function

f\_Protected(); // OK, f\_Protected is Derived\_Private's private function

//f\_Private(); // ERROR, f\_Private is Base's private function

}

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

private, protected and public keywords specifies different access control

from the derived class to the base class.

Access control:

1. None of the derived class can access anything that is private in Base.

2. Derived\_Public inherits public members of Base as public and the protected

members of Base as protected.

3. Derived\_Private inherits the public and protected members of Base as private.

4. Derived\_Protected inherits the public and protected members of Base as protected.

Casting point of view:

1. Anyone can cast a Derived\_Public\* to Base\*. Derived\_Public is a special kind of Base.

2. Derived\_Private's members and friends can cast a Derived\_Private\* to Base\*.

3. Derived\_Protected's member, friends and children can cast a Derived\_Protected\* to Base\*.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(void)

{

Derived\_Public d1;

d1.f\_Public(); // OK. f\_Public is Derived\_Public's public function.

Derived\_Protected d2;

//d2.f\_Public(); // ERROR. f\_Public is Derived\_Protected's protected function.

Base\* ptrBase = &d1;

//ptrBase = &d2; // ERROR!!!

return 0;

}

# 19 Advanced C++ - Maintain is-a Relation for Public Inheritance

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Maintain is-a Relation for Public Inheritance

Public inheritance -> is-a relation

A derived class should be able to do everything the base class can do.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma region Lesson 1

class dog

{

public:

virtual void bark(string msg = "just a")

{

cout << "Woof, I am " << msg << " dog." << endl;

}

};

class yellowdog : public dog

{

public:

virtual void bark(string msg = "a yellow")

{

cout << "Inside Yellow Dog!!!" << endl;

cout << "Woof, I am " << msg << " dog." << endl;

}

};

int main(void)

{

yellowdog\* ptrYellow = new yellowdog;

ptrYellow->bark();

dog\* ptrDog = ptrYellow;

// When ptrDog->bark is called, still we are calling the yellow

// dog's bark function(See the output). But the it will pickup the

// default value of message from Dog's bark function!!!

// Lesson: Never override default parameter value of virtual function!!!

ptrDog->bark();

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Inside Yellow Dog!!!

Woof, I am a yellow dog.

Inside Yellow Dog!!!

Woof, I am just a dog.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma endregion Lesson 1

#pragma region Lesson 2

class dog

{

public:

void bark(int age)

{

cout << "I am " << age << " years old" << endl;

}

virtual void bark(string msg = "just a")

{

cout << "Woof, I am " << msg << " dog." << endl;

}

};

class yellowdog : public dog

{

public:

using dog::bark;

virtual void bark(string msg = "a yellow")

{

cout << "Inside Yellow Dog!!!" << endl;

cout << "Woof, I am " << msg << " dog." << endl;

}

};

int main(void)

{

yellowdog\* ptrYellow = new yellowdog;

// Below code won't compile:

// Reason:

// When the compiler sees the bark function in the yellow dog, it

// stops REGARDLESS OF THE PARAMETERS. It won't look up the ladder.

// We have to use using dog::bark as shown above

ptrYellow->bark(5);

return 0;

}

#pragma endregion Lesson 2

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Summary:

1. Precise definition of classes.

2. Don't override non-virtual functions.

3. Don't override default parameter values for virtual functions.

4. Force inheritance of shadowed functions.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# 20 Advanced C++ - Understanding rvalue and lvalue

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Understanding rvalue and lvalue

Why do I care?

1. Understand C++ construct, and decipher compiler errors or warnings.

2. C++11 introduces rvalue reference.

What are they?

lvalue - An object that occupies some identifiable location in memory.

rvalue - An object that is not a lvalue.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class dog

{

public:

void bark(int age)

{

cout << "I am " << age << " years old" << endl;

}

virtual void bark(string msg = "just a")

{

cout << "Woof, I am " << msg << " dog." << endl;

}

};

int sum(int x, int y) { return x + y; }

int myGlobal = 0;

int& foo(void){ return myGlobal; }

int main(void)

{

{ // lvalue examples

int i; // i is a lvalue.

int\* ptr = &i;

dog d1; // lvalue of user defined type.

}

{ // rvalue examples

int x = 2; // 2 is an rvalue.

//x+2 = 8; // (i+2) is rvalue

}

{ // Reference (or lvalue reference)

int i;

int& r = i;

//int& y = 6; //ERROR!!!

//Exception is, const lvalue reference can be assigned a rvalue.

const int& rr = 5;

}

{ //Misconception 1: Functions or operator always yields rvalue

int x = 4 + 5;

int y = sum(x, 8);

foo() = 50; // function yielding lvalue!!!

int arr[4];

arr[2] = 4; // Operator [] almost always generates lvalue

}

{ //Misconception 2: lvalues are modifiable.

const int c = 1;

//c = 9; // ERROR!!!

}

{ //Misconception 3: rvalues are not modifiable.

dog().bark(); // bark() may change the state of dog!!!

}

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Summary:

1. Every C++ expression yields either an lvalue or an rvalue.

2. If the expression has an identifiable memory address, it's lvalue.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# 21 Advanced C++ - Static Polymorphism – Not there

# 22 Advanced C++ - Multiple Inheritance - Devil or Angel – Not there

# 23 Advanced C++ - Duality Of Public Inheritance - Interface & Implementation

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Duality Of Public Inheritance - Interface & Implementation

- Inheritance of interface

- Inheritance of implementation

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include <string>

using namespace std;

class Dog

{

public:

virtual void bark() = 0;

void run() { cout << "Woof, I am running." << endl; }

virtual void eat() { cout << "I am eating" << endl; } // impure virtual function

protected:

void sleep() { cout << "I am sleeping" << endl; }

};

class Yellowdog : public Dog

{

public:

// Interface only inheritance

virtual void bark() { cout << "Woof, I am yellow dog." << endl; }

// A non virtual function should not be overridden!!!!

// Yellow dog inherits interface of dog function as

// well as the implementation of run function.

// Since the function is protected, yellow dog inherits the function

// It only inherits the implementation but not the interface.

// To provide a service for sleep, it has to provide its own version.

void iSleep() { sleep(); }

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Summary:

1. Pure virtual public function - inherits interface only.

2. Non-virtual public function - inherits both interface and implementation.

3. Impure virtual public function - inherits interface and default implementation.

4. Protected functions - inherits implementation only.

Interface inheritance

1. Sub-typing.

2. Polymorphism.

Pitfall:

- Be careful of interface bloat.

- Interface do not reveal implementation.

Implementation inheritance:

1. Increase code complexity.

2. Not encouraged.

Guidelines of implementation inheritance

1. Do not use inheritance for code reuse, use composition.

2. Minimize the implementation in base classes. Base classes should be thin.

3. Minimize the level of hierarchies in implementation inheritance.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(void)

{

Yellowdog aDog;

aDog.bark();

return 0;

}

# 24 Advanced C++ - Code Reuse - Inheritance vs Composition

#pragma region ReuseWithInheritance

class Dog

{

public:

// Common activities

void bark(int age)

{

cout << "I am " << age << " years old" << endl;

}

};

class Bulldog : public Dog

{

public:

// Calls common activities to perform more tasks.

};

class ShepheredDog : public Dog

{

public:

// Calls common activities to perform more tasks.

};

int main(void)

{

return 0;

}

#pragma endregion ReuseWithInheritance

#pragma region ReuseWithComposition

class ActivityManager

{

// Common activities

void bark(int age)

{

cout << "I am " << age << " years old" << endl;

}

};

class Dog

{

public:

};

class Bulldog : public Dog

{

public:

ActivityManager\* ptrActMgr;

// Calls common activities to perform more tasks.

};

class ShepheredDog : public Dog

{

public:

ActivityManager\* ptrActMgr;

// Calls common activities to perform more tasks.

};

int main(void)

{

return 0;

}

#pragma endregion ReuseWithComposition

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Composition is better than inheritance

1. Less code coupling between reused code and re-user of the code

In case of inheritance....

a. Child class automatically inherits ALL parent class's public members.

b. Child class can access parent's protected members.

- Inheritance breaks encapsulation.

2. Dynamic binding

a. Inheritance is bound at compile time.

b. Composition can be bound either at compile time or at run time.

3. Flexible code construct

**Dog** **ActivityManager**

BullDog OutDoorActivityManager

ShepherdDog InDoorActivityManager

... ...

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# 25 Advanced C++ - Namespace and Keyword using

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Namespace and Keyword using

1. using directive : to bring all namespace members into current scope

Example: using namespace std;

\* using directive can only work with namespace.

2. using declaration

a. Bring one specific namespace member to current scope.

b. Bring a member from base class to current class's scope.

Example: using std::cout;

\* using declaration can work with class or with namespace.

\* using declaration for class can be using in class scope only!!!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class B

{

public:

int fun(int x)

{

using namespace std; // Local scope

cout << "From B" << endl;

return x;

}

};

class D :private B

{

public:

void g()

{

using namespace std; // Local scope

cout << "From D" << endl;

}

void h()

{

using std::cout;

using std::endl;

cout << "From D" << endl;

}

void fun(void)

{

using std::cout;

using std::endl;

cout << "From D" << endl;

}

using B::fun; // Class scope

//using namespace std; // Illegal

//using std::cout; // Illegal

};

namespace //Anonymous namespace

{

void h() { std::cout << "From h\n"; }

}

int main(void)

{

D d;

//d.fun(8); Initially this will not compile because of name hiding

d.fun(9); // After using B::fun; this will compile

// Can only be called within this file. Similar to global static h()

h();

return 0;

}

# 26 Advanced C++ - Koenig Lookup - Argument Dependent Lookup (ADL)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Koenig Lookup - Argument Dependent Lookup (ADL)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma region Example 1

namespace A

{

struct X {};

void g(X){ cout << "Calling A::g()" << endl; }

}

int main(void)

{

A::X x1;

// Even though we have not used the A::g(), compiler will look in

// the namespace where the argument type is located. In our case the

// argument X is located in the namespace A. This is called Koenig lookup

// or Argument Dependent Lookup (ADL)

g(x1);

return 0;

}

#pragma endregion Example 1

#pragma region Example 2

class C

{

public:

struct Y{ };

static void h(Y) { cout << "Calling C::h()" << endl; }

};

int main(void)

{

C::Y y;

//h(y); // This will not lookup in class. This will be an error

return 0;

}

#pragma endregion Example 2

#pragma region Example 3

namespace A

{

struct X {};

void g(X){ cout << "Calling A::g()" << endl; }

}

namespace C

{

void g(A::X) { cout << "Calling C::g()" << endl; }

void j()

{

A::X x;

//g(x); // Error -ambiguous call to overloaded function

// This is because of Koenig lookup, compiler can see 2 g() functions

}

}

int main(void)

{

C::j();

return 0;

}

#pragma endregion Example 3

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Summary:

With namespaces:

Compiler will search like ---> current scope -> next enclosed scope -> ... -> global scope

With classes

Compiler will search like ---> current class scope -> parent scope -> ... -> global scope

Name hiding will happen when higher priority scope defines a function with

same name in the lower priority scope.

To override the sequence:

1. Qualifier or using declaration.

2. Koenig lookup.

with classes

- Qualifier or using declaration.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/