Bitwise AND -> &



Bitwise OR -> |

Bitwise NOT -> ~

Bitwise XOR -> ^

Bitwise left shift <<

Bitwise right shift >>

Cin.ignore()

Cin.clear()

Reference variable must be initialized

A reference variable cannot change its association

String name = “sdf”;

String &refvar = name;

POINTER

Int \*p; // this a declaration of a pointer

\*p = 5 // the value of the variable which the pointer is pointed, will change

Int \* const r = &var; // it is possible to change the value of the variable

\*r = 20; // this is correct

const int \* p\_2 = &a; // this pointer cannot change the value that is under address its pointing

const int \* const p\_3 = &a; // this cannot change anything



ARRAYS

the name of an array is a pointer

int \*p = new int[10];

p[1]; // this sentence is going to get the value in position 1

const char \*a = “this is correct”;

char b[] = “this is correct too”;

RANDOM

srand(); //seed

time(NULL); // time in seconds

srand(time(NULL));

rand() % 49; // [0, 49]

rand() % 49 + 2; // [2, 49]

STRUCTURES

struct myobj {

string name

};

myobj newOb; // this is a definition

myObj testArray[4];

testArray[0].name = “this is correct”;

testArray->name = “this is correct cause the pointer is poiting the first element in the array”;

(\*&person[0]).name = “this is correct cause it is accessing the value in the address”;

struct myobj {

string name

} a, b; // it is defining two objects which are object “a” and object “b”

CLASSES

// a structure cannot have functions or methods instead of Classes

// By default all attributes in a class are privates

// it is necessary to define a constructor and destructor

class test {

private:

static int id;

public:

test();

~test();

};

test::test() {

this->id++;

}

// the variable will be the same and every instance will have different value

const test newObjec; // it is correct but it needs to access to its methods/functions it will be / //necessary to add the reserved word const after the definition and declaration function

void newFuntion() const;

FRIENDSHIP IN CLASSES

A class can access to private attributes of another class if they are friends

class A {

int send;

};

class B {

friend class A;

int des;

};

COPY CONSTRUCTOR

For do that is necessary to overload the constructor

classT newObjA;

classT newObjB;

newObjA = newObjB; // this Is not wrong but newObjA is pointing to newObjB

classT::classT(const classT & obj) {

this->attrA = obj.attrA;

}

OVERLOAD OPERATORS

class::operator int (){

return this->attrInt; // this is an int property

}

classT re;

int a = (int) re;

int class::operator+=(classT aux) {

this->attrInt = this->attrInt + aux.attrInt;

return this->attrInt;

}

CONSTRUCTOR TIP

classT(int = 0); // this definition means that the default value will be zero 0 if the constructor does not receive anything

INHERITANCE

These are an example of inheritance

class base {

protected:

int x; // this attribute will be inheritance for child classes

public:

int getX();

}

class child : public base { // it specifies that the methods and functions will have the same access

private:

int y;

public:

int getY();

int getX();

}

child::getY() {

this->x = 0; // this attribute was inherintaced

this->y =0;

}

child newChild;

newChild.getX();

newChild.base::getX(); // this is useful when it is necessary to access to the base method

VIRTUAL METHODS



virtual string getVoice(); // this an example

// the compiler always is going to look for a definition in a child class



ABSTRAC CLASS

// An abstract class cannot be instanced, and it can be identified by its methods

virtual string getVoice() = 0;



FUNCTION TEMPLATES

template<typename T, typename T2>

T add(T var1, T2 var2) {

return var1 +var2;

}

cout << add(2, 2.5) << endl; // the result will be a double, but if it needs to specify the type return value

cout << add<double, double>(2, 5.6) << endl; the result will be double

CLASS TEMPLATES

template<typename T>

class Point {

protected:

T x;

public:

T getX();

};

template<typename T>

Point<T>::getX(){

return this->z

}

// this an instance

Point<int> newPoint;

// this is an axample of inheritance

template<typename T>

class Point2D : public Point<T> {

…

}

EXCEPTIONS

catch(…){

//this is for everything

}

NAMESPACE



VECTOR

This type of array is useful when you know the total of elements that the system is going to handle during the execution time because when it adds a new element, the vector multiplies its size by two

std::vector<int> newVector;

AUTO (RESERVED WORD)

The word “auto” identifies the typo of variable the was assigned.

std::vector<int>::iterator it = currentVector.begin();

auto it = currentVector.begin();

The lines below executes the same;