C++ & Object Oriented Programming

by Merve Asiler

Class & Object

- **Class:** A data structure that keeps variables and methods inside.
- **Object:** An instance of a class.

You can consider int, float, double, char, string, etc. as the built-in data types whereas classes are the data types made by you.

```
int a;
ClassName b;
```

Sample Class Structure

```
class ClassName {

    [access modifier]

        <type> member1;

        <type> member2;

    [access modifier]

        <type> member3;

        <type> member4;

    …

} [instance list];
```

```
class Circle {

    private:

        int radius;

        float center[3];

    public:

        void setRadius(float r) {…};

        float computeArea() {…};

} mycircle, a, b, new_circle;
```

• You can also give the instance list separately:

```
Circle mycircle, a, b, new_circle;
```

Access to Class Members

- All the variables and functions (methods) are called as **member** of the class. They can be either public, private or protected (<u>Access modifiers</u>).

➢ **Public:** Member can be accessed from everywhere, like from other class instances, or from any function, etc.

➢ **Private:** Member can be accessed only from its own members or from the members of *friend* classes (*friendship* will be explained later). In default, all members are private.

➢ **Protected:** This access type is about *inheritance*, so let's explain it later.

:: Operator

When you want to define the methods of a class outside the class declaration, you need to use ':: ' operator.

```
class Circle {

  private:

      int radius;

      float center[3];

  public:

      void setRadius(float r);

};
```

Function declaration

Function definition

```
void Circle::setRadius(float r) {
        radius = r;
}
```

If you define just as below:
```
    void setRadius(float r) {
        float radius;
        radius = r;
    }
```

then it becomes another function other than Circle's setRadius() method.

. and -> Operators

- When you want to use a public member of an object via itself (or via pointer), then you need to use '**.**' (' -> ', respectively) operator.

```
class Circle {
    private:
        int radius;
        float center[3];
    public:
        void setRadius(float r);
};
```

```
int main() {
        Circle c;
        c.setRadius(100);
        Circle* d = new Circle();
        d->setRadius(150);
        delete d;
        return 0;
}
```

Constructors/Destructors

Constructors	Destructors
Special methods that construct an instance of the class.	Special methods that destroy an instance of the class.
Generally, **in constructors**, operations such as variable initialization and dynamic memory assignment are done.	If there is a dynamic allocation for some variable of the class, then it needs to be deallocated **in destructors**.
	Dynamic allocation: the spaces obtained by new or malloc/calloc/realloc keywords.

Constructors/Destructors

Take attention to declaration of constructor and destructor:

```cpp
class Polynomial {
  int degree;
  double *coeffs;

  public:
      Polynomial() {          // constructor
              degree = 0;
              coeffs = new double[1];
              coeffs[0] = 0;
      }

      ~Polynomial() {          // destructor
              delete[] coeffs;
      }
};
```

Constructor Properties

➢Constructors are functions that get called when a class is instantiated.

➢Constructors have the same name as their class.

➢Constructors may not pass off control to another constructor.

➢A default (zero argument) constructor is provided when no other constructors declared.

Constructor Properties (cont'd)

➢A constructor can be overloaded which means there can be more than one constructor of a class.

➢If overloaded, each constructor should have a distinct parameter list to uniquely determine the which constructor is called when an object is being instantiated.

➢The compiler assumes a default constructor unless you define one. The default constructor does not take any parameters (it is a *nullary* constructor).

Destructor Properties

➤ Destructors are methods that get called automatically when a class instance is destroyed.

   (i) If the object is not dynamically allocated (i.e. if held in stack), then destructor is called when the local scope of its existence has finished (e.g. At the end of a function).

   (ii) If the object is dynamically allocated, then destructor is called when delete operator is called.

➤ Destructors have no return type and have the same name as their class with a "~" prepended.

Destructor Properties (cont'd)

➢If no destructor is defined, a default destructor, which calls the destructor for all elements is provided by compiler. Note that default constructor does not deallocate the dynamically allocated members.

➢You may need to use destructor when

(i) the object assigns dynamic memory during its lifetime. In this case, allocated memory should be released to avoid memory leaks
(ii) the object opens resources such as files. Resources should be properly released, otherwise data loss may occur.

Constructors/Destructors

```
Class A {
    int var1;
    float var2;
    public:
        A() {};
        A(int v1, float v2) : var1(v1), var2(v2) {};
}
Class B {
    A var1;
    A* var2;
    public:
        B()  {var2 = new A(0, 1.0);}
        ~B() {delete var2;}
}
```

Copy Constructor

- You can also construct an object by copying from another object of the same type. In this case copy constructor is called automatically.

```cpp
class Polynomial {
  int degree;
  double *coeffs;

 public:
      Polynomial() {    // constructor
            degree = 0;
            coeffs = new double[1];
            coeffs[0] = 0;
      }

      ~Polynomial() {   // destructor
            delete[] coeffs;
      }
```

```cpp
      Polynomial(const Polynomial &p) {
            degree = p.degree;
            coeffs = new double[degree];
            memcpy(coeffs, p.coeffs,
                       sizeof(double) * degree);
      }
};
```

Default copy constructor also exists yet it just directly   assigns the values without any dynamic allocation, or anything else that you may need.

Assignment Operator =

- Assignment operator copies values from one instance to the other too, yet **the difference from copy constructor is** that the object which takes the copied values exist before the assignment operation. Therefore, you should clean the previous values of the object and then assign the new values.

```cpp
Polynomial & operator=(Polynomial & p) {
    if(this != &p) {          //Prevents self-assignment
        delete[] coeffs;
        degree = p.degree;
        coeffs = new double[degree];
        memcpy(coeffs, p.coeffs, sizeof(double) * degree);
    }
    return *this;
}
```

Canonical Form

All classes have each of the following:

- Default constructor

- Copy constructor

- Assignment operator

- Destructor

If the object instances manage dynamic memory or resources, you will have to override the default methods.

Copy operations handled with dynamic memory management is called **deep copy** whereas the others called **shallow copy**.

Call by Reference: & Operator

- Reference operator '**&**' is used either in <u>parameters</u> or <u>return types</u> of functions and means the object in the argument or returned will be the **instance itself, not its copy**.

```
Polynomial function1(Polynomial & p) {…}
Polynomial & function2(Polynomial p) {…}
Polynomial & function3(Polynomial & p) {…}
```

Copy Constructor vs. Assignment Operator
Pass by Value vs. Pass by Reference

```
void f(Polynomial p) {…}
void g(Polynomial &p) {…}

int main(void) {
    vector<double> c;      c.push_back(1.1);     c.push_back(2.0);
    Polynomial r(c);
    Polynomial s(r);       // Construct s from r
    Polynomial p;
    p = s;                 // Assign s to p
    Polynomial q = p;      // Construct q from p
    ...
    f(s);                  // s is passed by value,
                           // it's copied with the copy constructor
    g(p);                  // p is passed by reference
}
```

'Static' Keyword

- Static members are not specific to object, but valid for whole class. That is a common variable for all instances of that class.

- Especially can be needed during shallow copy.

```cpp
class Polynomial {
     public:
             static int count;
             Polynomial()  { ... count = count + 1; ... }
             ~Polynomial() { ... count = count - 1; ... }
};
//Globally initialize the static member
```

'this' Keyword

- The keyword **this** represents a pointer to the object whose member function is being executed. It is a pointer to the object itself.

- "this" pointer is allowed in a member function

- It can be used to return pointer to object itself or to check if a parameter passed to a member function is the object itself, etc..

'const' Keyword

4 types of usage:

i.   void function1(**const** ClassName& x) {…}

ii.   **const** ClassName& function2() {…}

iii.   void function3() **const** {…}

iv.   **const** ClassName& x = function2();

'const' with pointers

```
The following is okay:
Int value{ 5 };
const int* ptr{ &value }; // ptr points to a "const int"
value = 6; // the value is non-const when accessed through a non-const identifier

The following is not okay:
int value{ 5 };
const int* ptr{ &value }; // ptr points to a "const int"
*ptr = 6; // ptr treats its value as const, so changing the value through ptr is not legal

Also, the following is okay:
int value1{ 5 };
const int *ptr{ &value1 }; // ptr points to a const int
int value2{ 6 };
ptr = &value2; // okay, ptr now points at some other const int
```

'const' with pointers

The following is not okay:
```cpp
int value1{ 5 };
int value2{ 6 };
int* const ptr{ &value1 }; // okay, the const pointer is initialized to the address of value1
ptr = &value2; // not okay, once initialized, a const pointer can not be changed.
```

The following is okay:
```cpp
int value{ 5 };
int* const ptr{ &value }; // ptr will always point to value
*ptr = 6; // allowed, since ptr points to a non-const int
```

Taken from https://www.learncpp.com/cpp-tutorial/pointers-and-const/

C++ <vector>

- Vectors are sequence containers representing arrays that can change in size.
- You can use vectors by adding #include <vector> to the beginning of your code.
- .push_back() is the most generally used method of vector class.
- Holding pointers inside the vectors is recommended because of the vector concept explained below.

```
vector<Type> vect;   /* will allocate the vector on the
                            stack, but elements on the heap */


vector<Type*> vect; /* will allocate the vector on the
                           stack, pointers on the heap, but
                           elements that are pointed could be
                           anywhere (on stack or heap) */


vector<Type> * vect;/* both the vector and the elements will
                           be on the heap */
```

Inheritance

Inheritance is a relationship between two or more classes.

Derived class inherites behaviour and attributes of pre-existing (base) classes.

**Intended to help reuse of existing code with little or modification**.

Classes that are derived from others inherit all the accessible members of the base class.

Inherited Class

Derived Class

Bring all the accessible members from above and may have new members also.
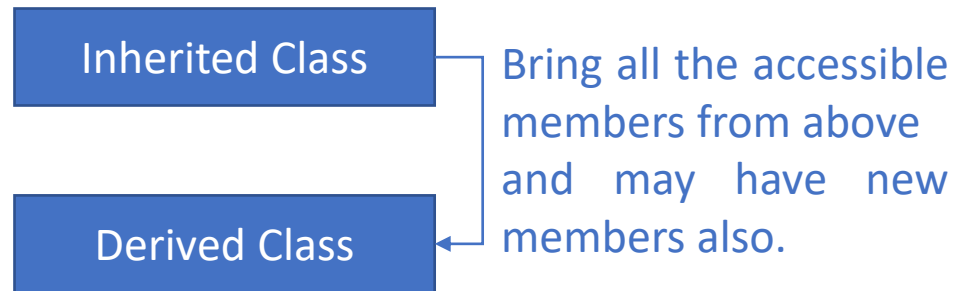
Inheritance

Example:

```
class House {
    protected:
        int number_of_rooms;
        int number_of_baths;
    public:
        void set(int nr, int nb) {…};
};

class Villa : public House {
        int number_of_flats;
};

class ApartmentFlat : public House {
        int flat_no;
        ApartmentFlat* neighbors;
    public:
        void addNeighbor(const ApartmentFlat& a) {…};
};
```
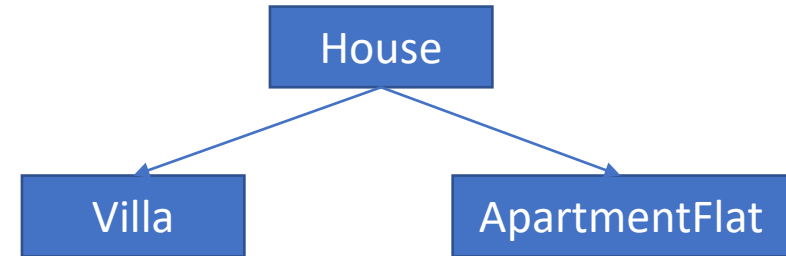

- See the ':' operator.

- New Access modifier: **protected**
  - ✓ Can be accessed by derived classes
  - x Can not be accessed by other classes

Inheritance

Important Properties!

➢ In principle, a derived class inherits every member of a base class except; its constructor/destructor, its =operator() members, and its friends.

➢ Although the constructors and destructors of the base class are not inherited themselves, its default constructor and destructor are always called when a new object of a derived class is created or destroyed.

➢ If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class:
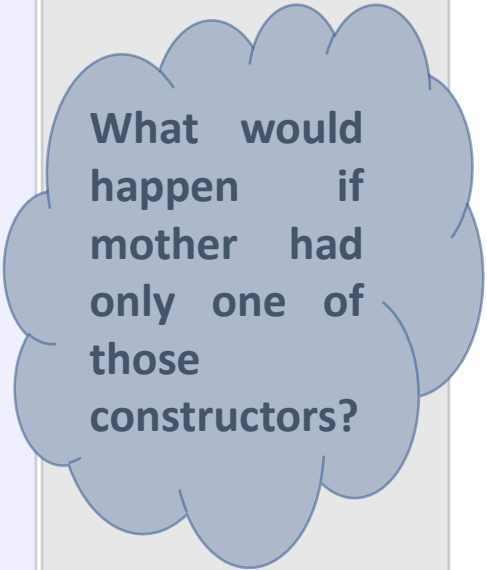
```
derived_constructor_name (parameters) : base_constructor_name (parameters) {…}
```

Inheritance

```cpp
1  // constructors and derived classes
2  #include <iostream>
3  using namespace std;
4
5  class mother {
6    public:
7      mother ()
8        { cout << "mother: no parameters\n"; }
9      mother (int a)
10       { cout << "mother: int parameter\n"; }
11 };
12
13 class daughter : public mother {
14   public:
15     daughter (int a)
16       { cout << "daughter: int parameter\n\n"; }
17 };
18
19 class son : public mother {
20   public:
21     son (int a) : mother (a)
22       { cout << "son: int parameter\n\n"; }
23 };
24
25 int main () {
26   daughter cynthia (0);
27   son daniel(0);
28
29   return 0;
30 }
```

```
mother: no parameters
daughter: int parameter

mother: int parameter
son: int parameter
```

What would happen if mother had only one of those constructors?

Inheritance

Inheritance type:

The inherited members will have access permission of at least the specified modifier given in the declaration:

```
derived_constructor_name (parameters) :
public/protected/private base_constructor_name
(parameters) {…}
```

Multiple Inheritance

A class can be derived and carry properties form multiple classes just by separating the base classes with comma in the declaration.

Polymorphism

- **C++ polymorphism** means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

```
class Virus {…};

class A : public Virus {…};

Virus* v;

A a;

v = &a;        /* referencing object of type A with
                   pointer of type Virus.
                   Thanks to polymorphism ☺ */
```

Virtual Functions

Virtual member functions
- Accessed by indirection through pointer in object.
- May be redefined in derived subclasses.
- The exact function to call determined dynamically
- Non-virtual functions are ordinary functions.
- Cannot redefine in subclasses (but can overload).
- Member functions are virtual if explicitly declared or inherited as virtual, else non-virtual

Virtual Functions

```cpp
class parent {
    public:
        void printclass() {
            printf("p ");
        };
        virtual void printvirtual() {
            printf("p ");
        };
};
class child : public parent {
    public:
        void printclass() {
            printf("c ");
        };
        void printOwnFunction() {
            printf("c ");
        };
        virtual void printvirtual() {
            printf("c ");
        };
};
```

```cpp
int main() {
    parent p; child c; parent *q;
    p.printclass();
    p.printvirtual();

    c.printclass();
    c.printvirtual();
    c.printOwnFunction();

    q = &p;
    q->printclass();
    q->printvirtual();

    q = &c;
    q->printclass();
    q->printvirtual();
    dynamic_cast<child*>(q)->printOwnFunction();

    return 0;
}                   // output: p p c c c p p p c c
```

Abstraction

- Abstract class is a class which you cannot instantiate.

- An abstract class is a class that is designed to be specifically used as a base class.

- An abstract class contains at least one abstract method which is a pure virtual function.

- You declare a pure virtual function by using a pure specifier (= 0) in the declaration of a virtual member function in the class declaration.

- These methods forces derived classes to implement those methods.

Abstraction

```cpp
class Virus {
    protected:
        int rna_length;
            string RNA;
    public:
        Virus(int length, string rna) {
            rna_length = length;
            RNA = rna;
        }

        void printRNA() const {
            cout << RNA << endl;
        }
                                    abstraction

        virtual void mutate() = 0;
};
```

```cpp
class A : public Virus {
    public:
        A(int length, string rna) :
                        Virus(length, rna) {};

        void mutate() {…}
};

class B : public Virus {
    public:
        A(int length, string rna) :
                        Virus(length, rna) {};

        void mutate() {…}
};
```

Templates

- **Templates** are a feature of the C++ programming language that allow functions and classes to operate with generic types.

- It allows a function or class to work on many different data types without being rewritten for each one.

Templates

```
1  // template specialization
2  #include <iostream>
3  using namespace std;
4
5  // class template:
6  template <class T>
7  class mycontainer {
8      T element;
9    public:
10     mycontainer (T arg) {element=arg;}
11     T increase () {return ++element;}
12 };
13
14 // class template specialization:
15 template <>
16 class mycontainer <char> {
17     char element;
18   public:
19     mycontainer (char arg) {element=arg;}
20     char uppercase ()
21     {
22       if ((element>='a')&&(element<='z'))
23       element+='A'-'a';
24       return element;
25     }
26 };
```

```
28 int main () {
29   mycontainer<int> myint (7);
30   mycontainer<char> mychar ('j');
31   cout << myint.increase() << endl;
32   cout << mychar.uppercase() << endl;
33   return 0;
34 }
```

Output:
8
J

```
1  template <class T> class mycontainer { ... };
2  template <> class mycontainer <char> { ... };
```

'friend' Keyword

- If a class A is introduced to another class B as 'friend', then B gives permission to A for accessing any private/protected members.

- Instead of a class, only a single function can be introduced as friend also.

Syntax:

```
class A {
        B* b;
    public:
        void increaseNum() {
                b->num ++;
        };
};
```

```
class B {
        friend class A;
        int num;
    public:
        friend void func(B b);
};
void func(B b) {
        cout << b.num << endl;
}
```

Composition

➤ Relationship between objects, where one object *owns* or *has* the other object.

➤ For the example below, Car *has* or *owns* Motor.

- ▪ When Car is build, it's motor is built also.
- ▪ When Car is destroyed it's motor is destroyed.

```
class Car {

    private:

        Motor* motor;

    public:

        Car()  {motor = new Motor();}

        ~Car() { delete motor; }

};
```

Aggregation

➤ In composition, when the owning object is destroyed, so are the contained objects.

➤ In aggregation this is not necessarily true.

➤ Aggregation is a form of composition, where contained objects can still "live" after the owning object is released.

➤ For the example below, Departments has professors, if department is closed, professors still live...

```
class Department {
    private:
        Professor* professor;
    public:
        Department(Professor* prof):professor(prof) { }
        ~Department() { }
};
```

'typedef' Keyword

Typedef is used to create an alias to a type

```
typedef unsigned char byte;
byte mybyte;
unsigned char mybyte;
```

byte now represents an unsigned char

Both definitions of mybyte are equivalent to the compiler.

Header Guards

- Header Guards are a common C/C++ idiom
- Wrap each header file with the lines:

```
#ifndef FILENAME_H
#define FILENAME_H
<header file body here>
#endif /* FILENAME_H */
```

- Include guards are used to prevent multiple declarations of same names.

C Preprocessor

C/C++ files are preprocessed with CPP (C Preprocessor) by default before compilation.

CPP processes files line by line and only lines starting with `#` are handled .

- `#define` lets you define macros or constants, or it simply adds a name into CPP's symbol table
- `#include` copies the contents of a file to the point of inclusion.
- `#ifdef` name and `#ifndef` name lets you test the existence of a name. If name is existing, the portion until `#endif` is included in the result, otherwise it is discarded. You can see that include guards simply prevent multiple declarations of same variables.

Compilation

- If you want to compile source1.cpp, source2.cpp, … sourceN.cpp, the comand below works:

  ```
  g++ -std=c++11 executable_name source1.cpp source2.cpp …
  sourceN.cpp
  ```

- Then you can run by

  ```
  ./executable_name
  ```

  command where `executable_name` is any name that you give.

- If you want to see the valgrind results for memory leak checking, after the compilation you need to run your executable as below:

  ```
  valgrind --leak-check=full ./executable_name
  ```

Do not forget to install valgrind `sudo apt-get install valgrind` of course ☺

Makefile

```makefile
CPP=g++            # g++ -std=c++11   for compiling with c++11

CFLAGS= -c -Wall
all: exe_prog

exe_prog: source_file1.o source_file2.o
	$(CPP) -o exe_prog source_file1.o source_file2.o
source_file1.o: source_file1.cpp
	$(CPP) $(CFLAGS) source_file1.cpp
source_file2.o: source_file2.cpp
	$(CPP) $(CFLAGS) source_file2.cpp
valgrind:       # checking the existence of memory leaks
	valgrind --leak-check=full ./exe_prog
clean:
	rm -rf *o exe_prog
```

Valgrind Output

Valgrind output showing the existence of memory leaks

Valgrind Output


```
osboxes@osboxes:~/Desktop/Examples/ex4$ valgrind --leak-check=full ./main
==3763== Memcheck, a memory error detector
==3763== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3763== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==3763== Command: ./main
==3763==
abstract base class' constructor called!
A's constructor called!
abstract base class' constructor called!
B's constructor called!
abstract base class' constructor called!
A's constructor called!
abstact base class' destructor called!
AUG
SSAUAGUG
abstact base class' destructor called!
abstact base class' destructor called!
==3763==
==3763== HEAP SUMMARY:
==3763==     in use at exit: 0 bytes in 0 blocks
==3763==   total heap usage: 7 allocs, 7 frees, 73,896 bytes allocated
==3763==
==3763== All heap blocks were freed -- no leaks are possible
==3763==
==3763== For counts of detected and suppressed errors, rerun with: -v
==3763== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Valgrind output showing there does not exist any memory leak

Common Compilation Options

- -Wall enable all warnings

- -ansi turn on strict ansi compliance

- -v verbose mode on. (lists which directories were searched for include files and libraries)

- -g generate debugging symbols

- -c compile only (create object file)

- -E run the preprocessor only

- -s generate assembly code (instead of object code)

- -I<dir> prepend directory to include path

- -L<dir> prepend directory to library path

- -l<libraryname> library to link (looks for lib<libraryname>.so, or .a, or .la) in library path

- -o<filename> specifies output filename

- -O[n] specifies optimization level (default 0)

- -D<name>[=value] define the give preprocessor Tag

QUESTIONS?