COMP3016

OOP with C++ Part 2

Immersive Games Technologies

Dr Ji-Jian Chin

University of Plymouth 2024

Recap of Last Week

- Implement classes in C++.
- Define access control over an object's members using friend, public, private and/or protected.
- Call an object's member function with the dot and scope operators.
- Define the use of constructors (default) and destructors for a class.
- Differentiate between static, const and mutable effects when applied to members of a class.
- Declare class access functions
- Make use of anonymous variables and classes
- Pass objects to methods through value, reference or address
- Control method behaviour with const.
- Use static to declare members and methods of a class
- Use new, new[], delete and delete[] operators to dynamically allocate memory to variables
- Avoid memory leaks by identifying how dynamically allocated memory links can be broken without de-allocation
- Use the hidden this pointer of object classes
- Declare friend methods and classes
- Implement composite and aggregate relationships between classes

This week

- File processing
- Inheritance

Topics for this lecture segment A

- Understand C++ concepts in file processing functions of:
- File Open
- File Read
- File Write
- File Close
- Apply the aforementioned C++ File processing functions

Input / Output with files

- The C++ language provides the following classes to perform file processing.
- These classes are:
  - ofstream: Stream class to write on files
  - ifstream: Stream class to read from files
  - fstream: Stream class to both read and write from/to files

- The classes above are derived directly from istream or ostream classes.
- Recall that cin is an object of class istream and cout is an object of class ostream.
- The file stream classes can be applied in a similar manner as used for cin and cout, with the exception that the file stream classes are associated to physical files.

Input / Output with files

- The code example below describes the current method of using the ostream class to write to the console screen

```
// basic file operations
#include <iostream>
using namespace std;

int main ()
{
  cout << "Writing this to screen. \n";
  return 0;
}
```

Input / Output with files

- The code example below describes the method of using the ofstream class to create a file of name ‘example.txt’ and inserts a sentence into the file based on similar approach as applied with cout.

```cpp
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

Input / Output – Step by Step Approach

- The first operation generally performed on an object of one of these classes is to associate it to a real file.
- This procedure is known as to *open a file*. An open file is represented within a program by a stream object and any input or output operation performed on this stream object will be applied to the physical file associated to it.
- In order to open a file with a stream object, the member function open() is applied:
  - open (filename, mode); Where filename parameter is a null-terminated character sequence of type const char * representing the name of the file to be opened, and mode parameter is an optional parameter with a combination of the following flags:

Flag type	Operations
ios::in	Open for input operations
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set to any value, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file. This flag can only be used in streams open for output-only operations.
ios::trunc	If the file opened for output operations already existed before, its previous content is deleted and replaced by the new one.

Input / Output – Step by Step Approach

- All these flags can be combined using the bitwise operator OR (|). For example, if to open the file example.bin in binary mode, this can be achieved by the following call to member function open():

```
ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app |
ios::binary);
```

- Each one of the **open()** member functions of the classes **ofstream**, **ifstream** and **fstream** has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

Input / Output – Step by Step Approach

- For ifstream and ofstream classes, ios::in and ios::out are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the open() member function.

- The default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

- Since the first task that is performed on a file stream object is generally to open a file, these three classes include a **constructor** that automatically calls the **open()** member function and has the exact same parameters as this member. Therefore the previous **myfile object** could alternatively be declared as:

```
ofstream myfile ("example.bin", ios::out | ios::app |
ios::binary);
```

Input / Output – Step by Step Approach

- To check if a file stream was successful in opening a file, this is achieved by calling to the member function is_open() with no arguments. This member function returns a bool value of true in the case that indeed the stream object is associated with an open file, or false otherwise:

```
if (myfile.is_open()) { /* ok, proceed with output */ }
```

- When input and output operations on a file are completed, **the file must be closed** so that its resources become available again. In order to do that the stream's member function **close**() is called. This member function takes no parameters, and what it does is to flush the associated buffers and close the file:

```
myfile.close();
```

Input / Output – Text files

- These files are designed to store text and thus all values that is input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

Data output operations on text files are performed in the same way as operated with cout:

```
// writing on a text file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile ("example.txt");
  if (myfile.is_open())
  {
    myfile << "This is a line.\n";
    myfile << "This is another line.\n";
    myfile.close();
  }
  else cout << "Unable to open file";
  return 0;
}
```

Input / Output – Text files

Data input from a file can also be performed in the same way that as operated with cin (Option 1):

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  string line;
  ifstream myfile ("example.txt");
  if (myfile.is_open())
  {
    while (! myfile.eof() )
    {
      getline (myfile,line);
      cout << line << endl;
    }
    myfile.close();
  }

  else cout << "Unable to open file";

  return 0;
}
```

Input / Output – Text files

Data input from a file can also be performed in the same way that as operated with cin (Option 2):

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  string line;
  ifstream myfile ("example2.txt");
  if (myfile.is_open())
  {
    while (! myfile.eof() )
    {
      myfile >> line;
      cout << line << endl;
    }
    myfile.close();
  }

  else cout << "Unable to open file";

  return 0;
}
```

Input / Output – Checking State Flags

Member functions exist to check the state of a stream (all of them return a bool value):

- bad(): Returns true if a reading or writing operation fails. For example in the case that an attempt is made to write to a file that is not open for writing or if the device where write process takes place has no space left (insufficient disk space).
- fail(): Returns true in the same cases as bad(), but also in the case that a format error happens, like when an alphabetical character is extracted when an attempt is made to read an integer number.
- good(): It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true.
- eof(): Returns true if a file open for reading has reached the end. *(Refer to previous slide example for usage of this function)*

Input / Output – Get & Put Stream Pointers

- All I/O streams objects have, at least, one internal stream pointer:
- ifstream, like istream, has a pointer known as the get pointer that points to the element to be read in the next input operation.
- ofstream, like ostream, has a pointer known as the put pointer that points to the location where the next element has to be written.
- Finally, fstream, inherits both, the get and the put pointers, from iostream (which is itself derived from both istream and ostream).
- These internal stream pointers that point to the reading or writing locations within a stream can be manipulated using the following member functions:

Input / Output – Get & Put Stream Pointers

- tellg() return an integer value representing the current position of the get stream pointer.
- tellp() return an integer value representing the current position of the put stream pointer.
- seekg() and seekp(): These functions allow the change of position for the position of the get and put stream pointers. Both functions are overloaded with two different prototypes. These prototypes are:
  - seekg(position); // the get pointer is changed to the absolute position. position parameter is of type pos_type
  - seekp(position); // the put pointer is changed to the absolute position.

Input / Output – Get & Put Stream Pointers

direction param (seekdir)	Description
ios::beg	offset counted from the beginning of the stream
ios::cur	offset counted from the current position of the stream pointer
ios::end	offset counted from the end of the stream

The following example uses the member functions as earlier explained to obtain the size of a file:

```
// obtaining file size
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  long begin,end;
  ifstream myfile ("example.txt");
  begin = myfile.tellg();
  myfile.seekg (0, ios::end);
  end = myfile.tellg();
  myfile.close();
  cout << "size is: " << (end-begin) << " bytes.\n";
  return 0;
}
```

```
size is: 158 bytes.
```

Input / Output – Binary files

- In binary files, to input and output data with the extraction and insertion operators (<< and >>) and functions like getline is not efficient, since data formatting is not needed, and data may not use the separation codes used by text files to separate elements (like space, newline, etc…).
- File streams include two member functions specifically designed to input and output binary data sequentially: write and read.
- The first one (write) is a member function of ostream inherited by ofstream.
- The second one (read) is a member function of istream that is inherited by ifstream.
- Objects of class fstream have both members. Their prototypes are:
  - write (memory_block, size);
  - read (memory_block, size);
- memory_block parameter is of type "pointer to char" (char*), and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken.
- size parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.

Input / Output – Binary files

```
// reading a complete binary file
#include <iostream>
#include <fstream>
using namespace std;

ifstream::pos_type size;
char * memblock;

int main () {
  ifstream file ("example.bin", ios::in|ios::binary|ios::ate);
  if (file.is_open())
  {
    size = file.tellg();
    memblock = new char [size];
    file.seekg (0, ios::beg);
    file.read (memblock, size);
    file.close();

    cout << "the complete file content is in memory";

    delete[] memblock;
  }
  else
  { cout << "Unable to open file“ };
  return 0;
}
```

Input / Output – Binary files

- In the example from the previous slide, the entire file is read and stored in a memory block.
- First, the file is open with the ios::ate flag, which means that the get pointer will be <u>positioned at the end of the file</u>. This way, when tellg() is called, the size of the file will be obtained directly. Notice that variable size is declared as of type ifstream::pos_type
- ifstream::pos_type is a specific type used for buffer and file positioning and is the type returned by file.tellg(). This type is defined as an integer type and can safely be converted to another integer type large enough to contain the size of the file. For a file with a size under 2GB int could be used:

```
int size;
size = (int) file.tellg();
```

- Once size of the file has been obtained, dynamic memory block is allocated, large enough to hold the entire file:

```
memblock = new char[size];
```

Input / Output – Binary files

- After dynamic memory allocation, the get pointer is set at the beginning of the file *(note that initially that the file was opened with this pointer at the end)*,
- The entire file content is then read into the dynamic memory
- Finally, the file is closed.

```
file.seekg (0, ios::beg);
file.read (memblock, size);
file.close();
```

- Once the file content has been read into memory, further processing can be carried out to process the data in memory.
- The process of writing binary data into file is also similar in a reverse manner.

Break time

Topics for this lecture segment B

- Define inheritance and identify the different types available
- Create code for classes that show inheritance properties
- Differentiate between access specifiers for class members and inheritance specifiers class objects
- Define multiple inheritance and make use in program code

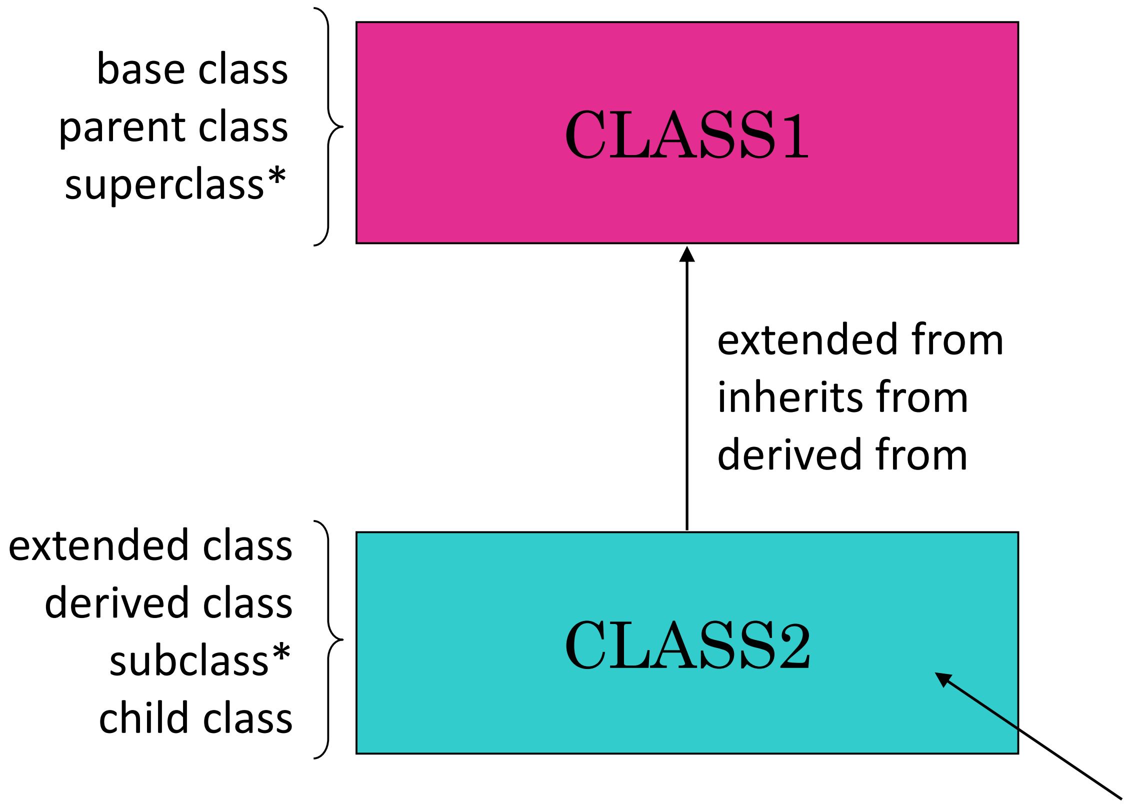
Class inheritance

- In C++, inheritance of objects is provided as a built-in mechanism by allowing a new class (subclass/derived class) to be built by deriving from an existing base class (super class, parent class).

- Inheritance promotes code reuse since portions of the code in the super class is inherited by the subclass and thus need not be rewritten

  - *Reducing the amount of code to be created*
  - *Preserving the correctness and consistency of the code*

Inheritance terminology

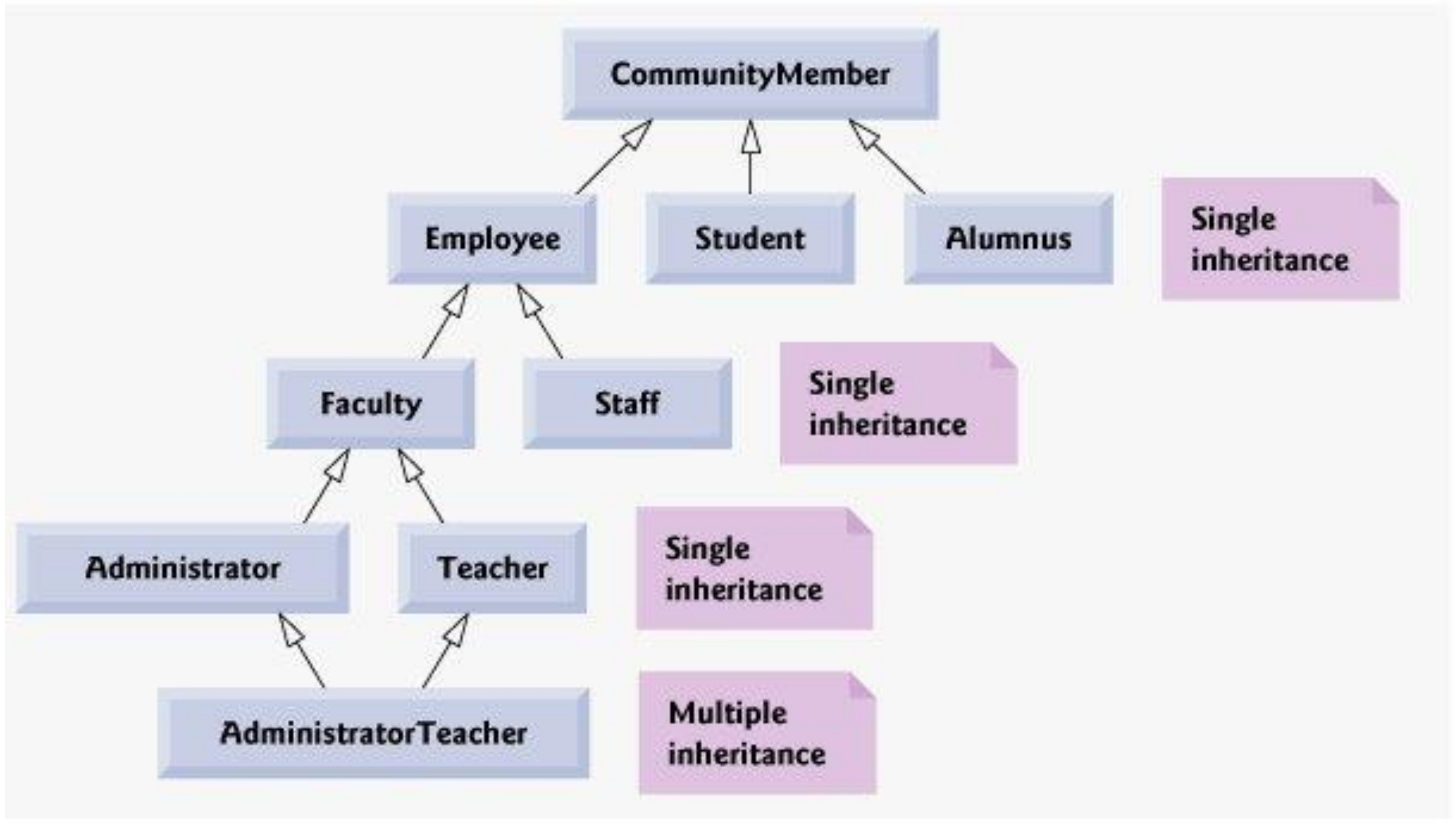


CLASS2 ends up with all the inheritable data fields and functions of CLASS1. It also adds its own new data fields and functions (i.e. specialization of the class)

Class inheritance hierarchy

- When implementing inheritance, the base class usually represents a more general form of an object and derived classes represent more specialized objects in that hierarchy.
- A derived class contains behaviors inherited from its base class plus additional behaviors - it can also customize existing behaviors from a class (function overloading/override)
- A direct base class is the base class from which a derived class explicitly (directly) inherits from.
- An indirect base class is inherited two or more levels up the hierarchy (i.e. super-superclass)
- If a class only inherits from a single base class, then it represents single inheritance.
- C++ supports multiple inheritance where a class inherits from multiple (possibly unrelated) base classes - complex and easy to create errors!

Sample Class inheritance hierarchy



Class inheritance example

- Assuming a class Student has already been defined and implemented and a new class GradStudent is needed.

- GradStudent is a specialized group of Student .

- Rather than building the class GradStudent from scratch, class Student could serve as the base building block, to which additional data and methods are added to form the new class GradStudent.

- The GradStudent class is said to have inherited from the class Student (GradStudent is a derived class or a subclass of class Student).The class Student is called the base class / superclass of class GradStudent .

- The derived class GradStudent inherits all of the data and methods from the base class Student (except for constructors, destructors, and an overload of the assignment operator) besides having its own unique set of data and methods.

Class inheritance types

- C++ provides three ways to support inheritance → based around how a class controls access to data and functions of a class - namely public, private and protected.
- With public inheritance, every object of a derived class is also an object of that class's base class.
- A derived class may access/interact with a base class's non-private data and functions only, even though they inherit ALL the contents of the base class.
- One problem with inheritance is that a derived class may inherit properties that it does not and/or should not need/have.

Public Class Inheritance Syntax

- Inheritance in C++ takes place between classes.
- When one class inherits from another, the derived class inherits the variables and functions of the base class.
- These variables and functions become part of the derived class.
- The syntax to tell the compiler to create a new class that inherits from another is

```
class <derivedclass> : public <baseclass> { ...
```

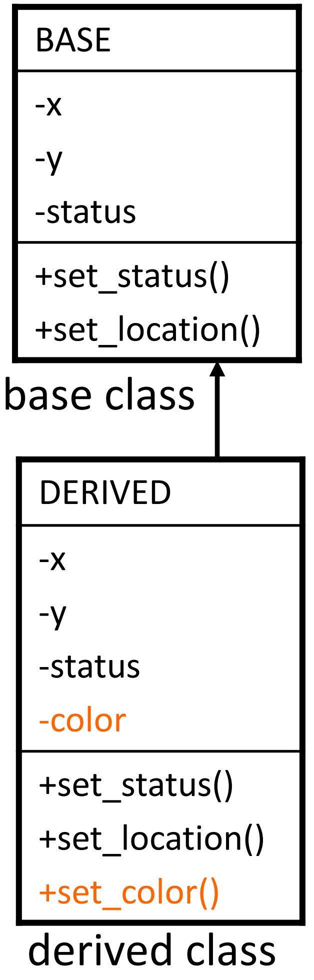
inheritance operator uses single colon

- Therefore, to derive a new class 'B' from an existing class 'A' -
- `class B : public A { ... };`
- The keyword `public` indicates that the derivation is public, i.e. the public members (data and methods) in the base class are also public in the derived class

- private members of a base class are not visible in the derived class even though they are inherited - they can only be accessed through a public member inherited from the base class that is visible by the derived class
- If a private member in the base class is to be accessed directly by the derived class, they must be made visible to the derived class by placing it under the access specifier protected in the base class.
- Without inheritance, a protected member behaves just like a private member and is only visible within the class.
- With public inheritance, a protected member in the base class is protected in the derived class, i.e., it is visible in the derived class (but not outside the class scope, i.e., in function main).
- A protected member is visible throughout the class hierarchy and may be accessed by all methods within the class hierarchy as well as friend functions
- Derived-class member functions can refer to public and protected members of the base class simply by using the member names.

```
class BASE {
public:
   void set_status( int );
   void set_location( int,
int );
private:
   int x;
   int y;
   int status;
};
```

```
class DERIVED : public BASE{
public:
   void set_color( int );
private:
   int color;
};
```

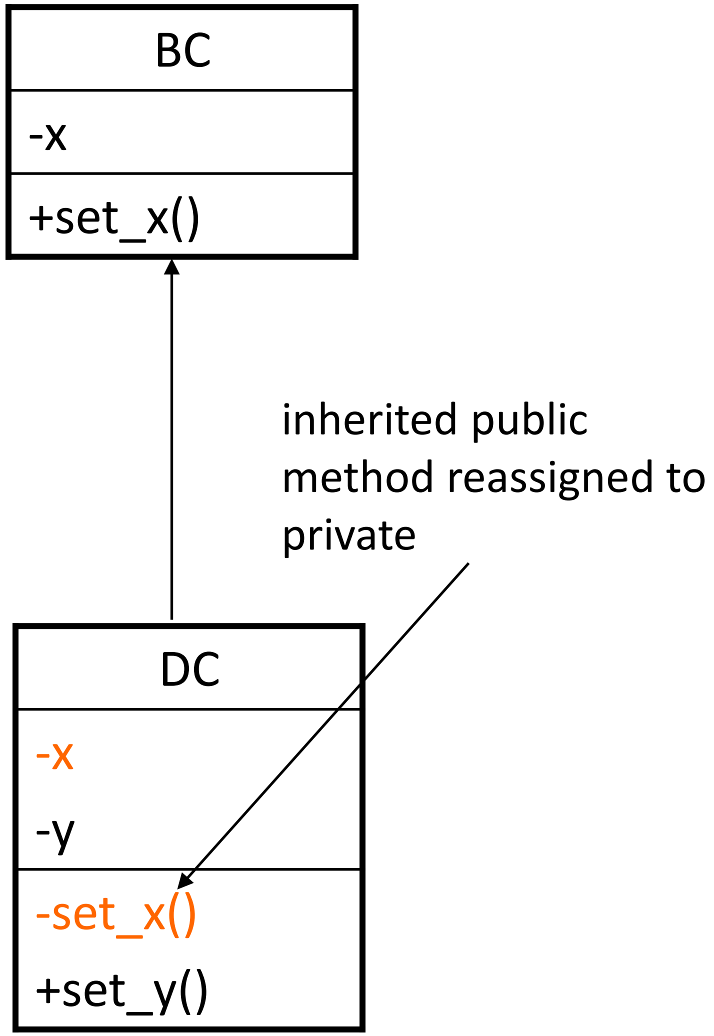


```
class BC { // base class
public:
    void set_x( int a ) { x = a; }
protected:
    int get_x( ) const { return x; }
private:
    int x;             };

class DC : public BC { // derived class
public:
    void add2( ) {
    int c = get_x( );
    set_x( c + 2 );    }
};

int main( ) {
  DC d;
  d.set_x( 3 );  // OK: set_x is public in DC
  cout << d.get_x( ) << endl;  // ERROR: get_x is protected
  d.x = 77;  // ERROR: x is private in BC
  d.add2( );  // OK: add2 is public in DC
  return 0;
}
```

Change inheritance type of members

- In certain situations, a `public` method in the base class may be irrelevant in the derived class.
- Therefore, the calling of this method by the objects of the derived class should be prohibited.
- Adjusting a "public" method in a base class to "private" status in a derived class is done with a `using` declaration that disables the usage of the method outside the derived class.
- Note that you can't change a private base class data member to a public one in the derived class → this goes against the concept of inheritance.

```
// base class
class BC {
public:
    void set_x( float a ) {
        x = a;
    }
private:
    float x;
};

// derived class
class DC : public BC {
public:
    void set_y( float b ) {
        y = b;
    }
private:
    float y;
    using BC::set_x;
};
```



Inherited class member overriding

- When a derived class adds a member (data or method) with the same name as the base class, the local member hides the inherited member (i.e. overrides original member).
- If the overridden member is a data type, then the local variable is used when the identified is referenced.
- If the overridden member is a method, then the local method of the derived class is called (even if they have different parameter list).
- Access to the inherited member from the base class is available with use of the scope operator (::) using the syntax

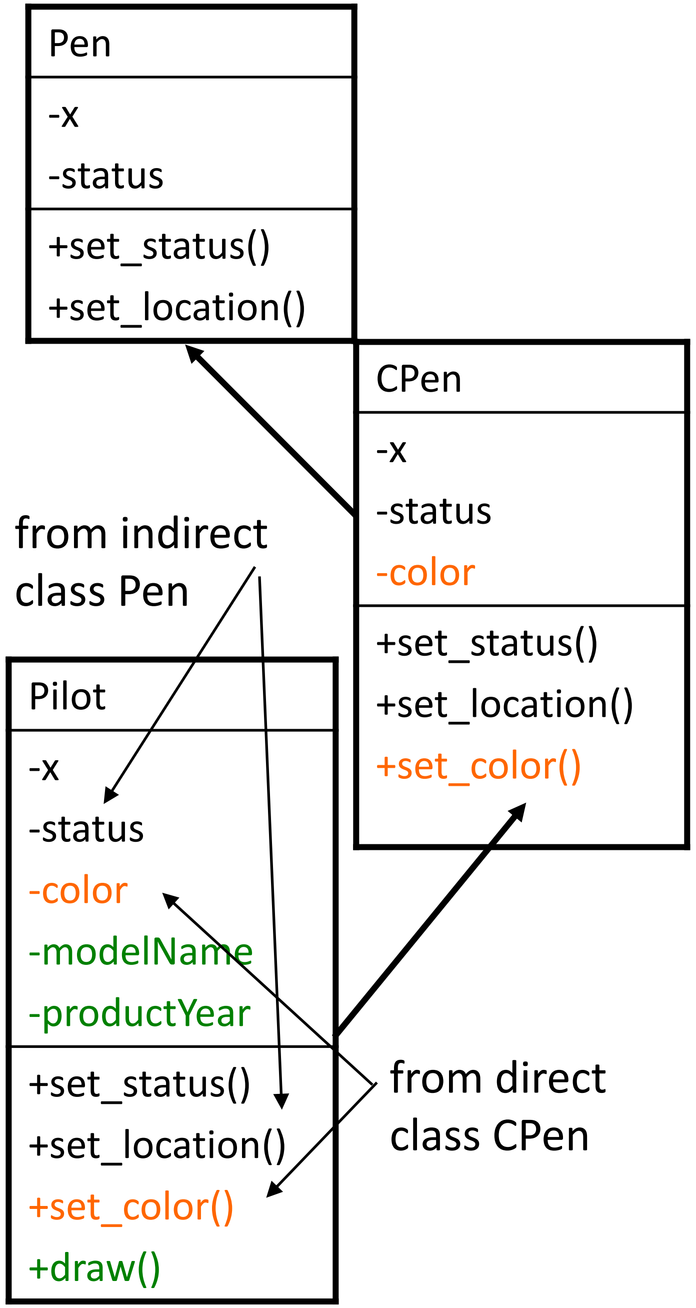
```
<objectvar>.<baseclass>::<member>
```

```
class BC {
public:
  void h( float );  // BC::h
};

class DC : public BC {
public:
  void h( char [ ] );  // overrides BC::h
};

int main( ) {
  DC d1;
  d1.h( “abc” );  // calls DC::h, not BC::h
  d1.h( 777.77 );  // ERROR: DC::h hides BC::h, floating-
             // point number not acceptable
  d1.BC::h( 777.77 ); // OK: invokes BC::h to receive a
             // floating-point number
  return 0;
}
```

Indirect inheritance

- Data members and member functions may traverse several inheritance links as they are included from a base class to a derived class.
- Inheritance may be either direct (to a derived class from a direct base class) or indirect (to a derived class from an indirect base class).

```
class Pen {
public:
    void set_status( int );
    void set_location( int, int );
private:
    int x;
    int status;
};

class CPen : public Pen {
public:
    void set_color( int );
private:
    int color;
};

class Pilot : public CPen {
public:
    void draw( );
private:
    char[10] modelName;
    int productYear;
};
```



Private inheritance

- If the keyword public is omitted, the inheritance is default to private:

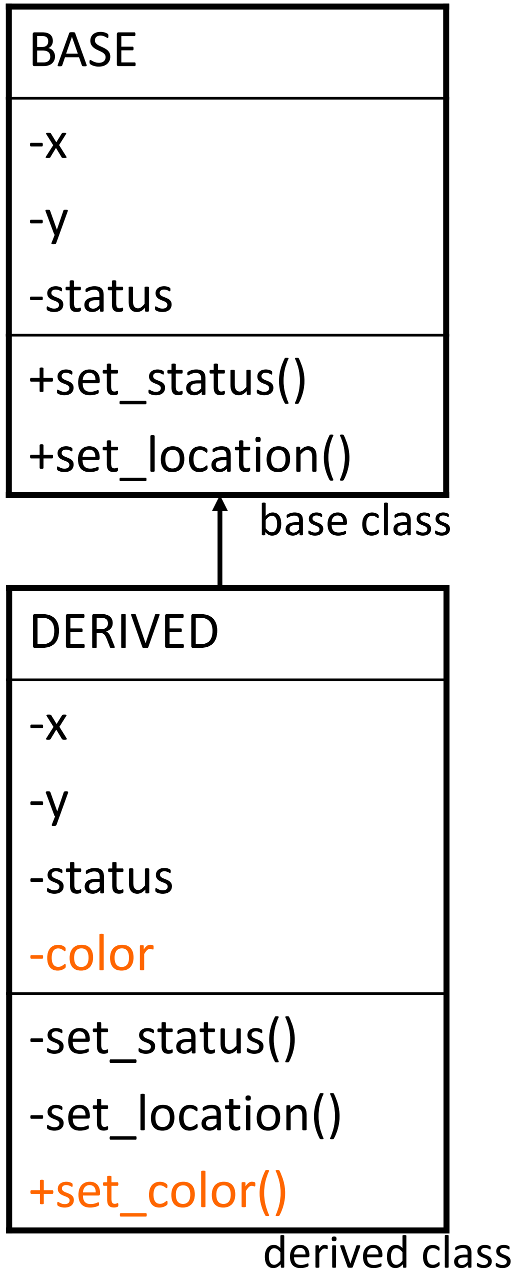
```
class DC : BC { // *** Caution: default inheritance is private
        // ...
};
or
class DC : private BC {    // *** private derived class
        // ...
};
```

- In a *private derivation*:
  - Each public member in the base class is private in the derived class
  - Each protected member in the base class is private in the derived class
  - Each private member in the base class is visible only in the base class

```
class BASE {
public:
   void set_status( int );
   void set_location( int, int );
private:
   int x;
   int y;
   int status;
};
```

```
class DERIVED : private Pen {
public:
   void set_color( int );
private:
   int color;
};
```

public members from base class are now private, so there is no way to access the private data members of the base class



Protected inheritance

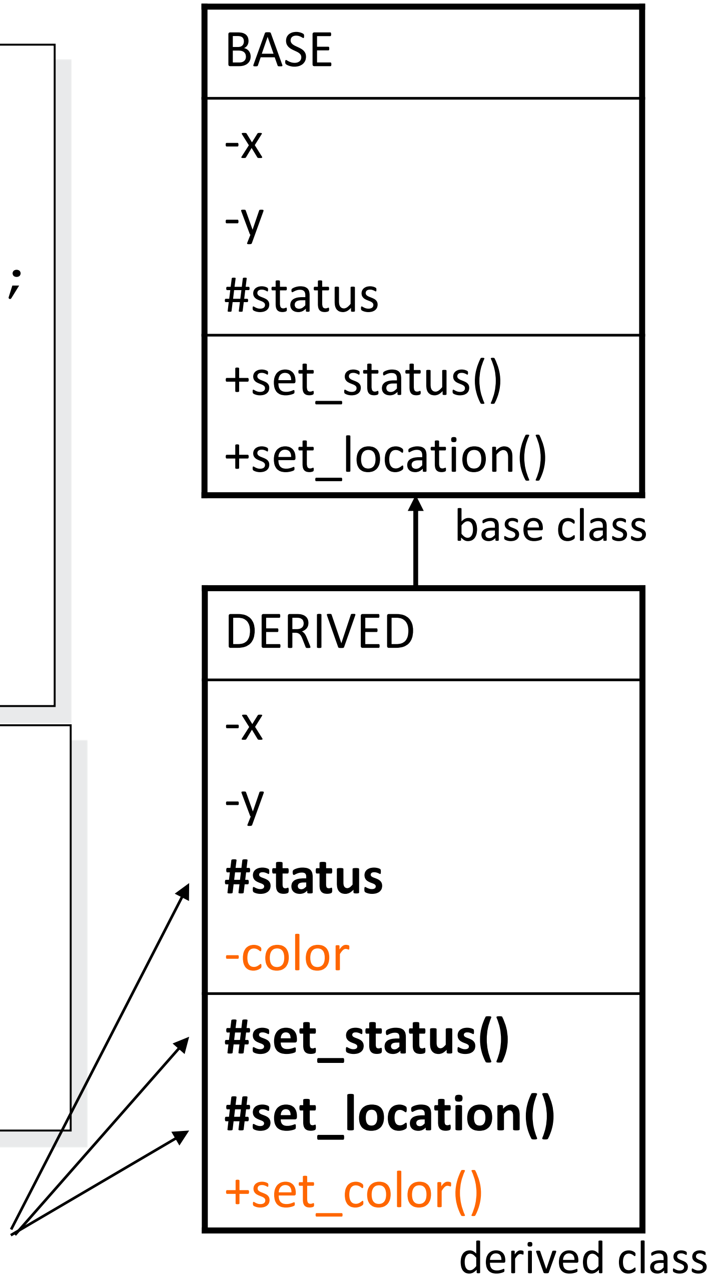
- The nature of a inheritance relationship can also be made protected by:

```
class DC : protected BC {          // *** protected derived class
        // …
};
```

- In a *protected derivation*:
  - Each public member in the base class is protected in the derived class
  - Each protected member in the base class is protected in the derived class
  - Each private member in the base class is visible only in the base class

```
class BASE {
public:
    void set_status( int );
    void set_location( int, int );
private:
    int x;
    int y;
protected:
    int status;
};
```

```
class DERIVED : protected Pen {
public:
    void set_color( int );
private:
    int color;
};
```



protected and public members are inherited as protected

Inherited constructors and destructors

- Each derived class has its own constructor and destructor (whether explicit or implicitly defined) BUT it does NOT inherit the constructor and destructor from the base class.
- A base class constructor (if any) is first invoked when a derived class object is created.
- The base class constructor handles initialization and other matters for the elements from the base class inherited by the derived class object.
- The derived class constructor (if any) is then invoked to handle the additional initialization required specifically within the derived class.
- Base class constructors are often sufficient for the derived class.
- A constructor specific to a derived class may invoke a base class constructor, if one exists.

- The derived class constructor invokes the base class constructor before executing its own body and its own constructor.
- If a base class has constructors but no default constructor (or has been disabled), a derived class constructor must explicitly invoke some base class constructor.
- Unless absolutely necessary, it is advisable to let a base class always have a default constructor.
- When a base class has a default constructor and that a derived class has constructors, the base class default constructor is invoked automatically whenever a derived class object is created, unless the derived class constructor explicitly called other available overloaded constructor.
- It is legal but unnecessary for a derived class constructor to invoke the base class default constructor explicitly.
- Under inheritance, the destructor is executed in the reverse order of constructor.
- The derived class destructor is executed first before the base class destructor.
- This ensures that the most recently allocated storage (by constructor) is the first storage to be freed
- Unlike constructors, a destructor never explicitly invokes another destructor.

```
class Animal {
public:
 Animal( ) { species = “Animal”;}
 Animal( const char* s ) { species = s; }
private:
 string species;
};
class Primate : public Animal {
 public:
       Primate( ) : Animal( “Primate” )
              { heart_cham = 2; }
       Primate( int n ) : Animal( “Primate” )
              { heart_cham = n; }
 private:
       int heart_cham;
};

int main( ) {
 Primate godzilla;
 Primate human( 4 );
 return 0;
}
```

Default constructor for base class

Parameterized constructor for base class

Default constructor for derived class calls parameterized base class constructor

Invokes derived class default constructor Primate(), which in turn invokes parameterized Animal() with argument “Primate”

Invokes derived class parameterized constructor Primate(int), which in turn invokes parameterized Animal() with argument “Primate”

Multiple Inheritance

- In multiple inheritance, the derived class has multiple base classes – each separated by commas during the declaration of the class.
- The rules of inheritance and access do not change from single inheritance to multiple inheritances.
- Multiple inheritances increase the risk of name conflict.
- Multiple inheritances sometime lead to the situation in which a derived class inherits multiple times from the same indirect base class.
- This is resolved with virtual base classes.

Multiple inheritance example 1

```
class Input { // provides input operations only
  ...
};
class Output { // provides output operations only
   ...
};
class InOutput : public Input, public Output {
  // provides both input and output operations
  ...
};
```

Multiple inheritance example 2

```
class IBC { // indirect base class
 int x;
};
class DBC1 : public IBC { //direct base class 1
};
class DBC2 : public IBC { //direct base class 2
};
class DC : public DBC1, public DBC2 {
  // DC inherits TWO copies of private int “x”
};
```

Resolving multiple inheritance conflict

```
class IBC { // indirect base class

  int x;

};

class DBC1 : virtual public IBC { //direct base class 1

};

class DBC2 : virtual public IBC { //direct base class 2

};

class DC : public DBC1, public DBC2 {

  // x is only inherited ONCE here

};
```

The keyword **virtual** in DBC1 and DBC2 tells DC to inherit only one copy of whatever DBC1 and DBC2 inherit from their common ancestor IBC

DC now makes use of *virtual* base classes. Consequently, DC is now responsible for creating/calling/constructing IBC for its own use

Pointing/referring to a base class

- Data types can be referenced directly, by reference or through pointers → this also applies to objects/classes.

- An object can be referenced directly,

```
Time timeobj1;
```

  or a pointer

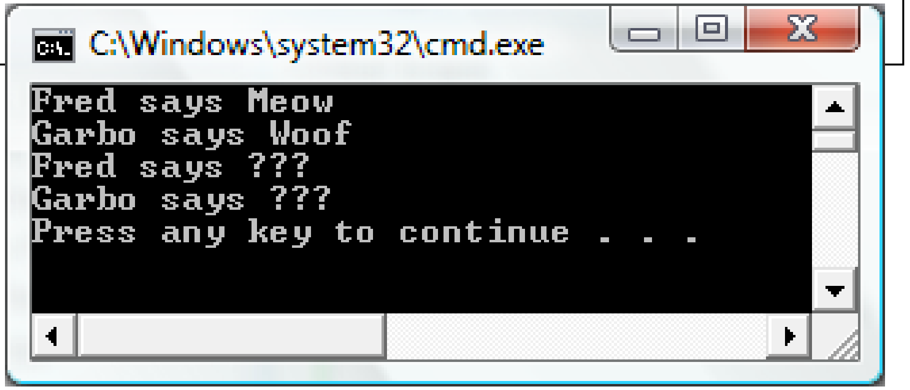
```
Time *timeptr1 = &timeobj1;
```

- The problem comes with classes that are derived from other classes - a pointer to the BASE class also works when pointed at a DERIVED class!

- Although the pointer *points* correctly, it can only access the BASE members of the DERIVED class - if there are overridden members in the DERIVED class, they won't be seen

```
class Animal {
protected:
    string m_strName;
    Animal(string strName) { m_strName =strName; }
public:
    string GetName() { return m_strName; }
    const string Speak() { return "???"; }
};

class Cat: public Animal {
public:
    Cat(string strName) : Animal(strName) {}
    const string Speak() { return "Meow"; }
};

class Dog: public Animal {
public:
    Dog(string strName): Animal(strName) {}
    const string Speak() { return "Woof"; }
};
```

```
int main() {
  Cat cCat("Fred");
  cout << cCat.GetName() << “ says " << cCat.Speak() << endl;

  Dog cDog("Garbo");
  cout << cDog.GetName() << “ says " << cDog.Speak() << endl;

  Animal *pAnimal1 = &cCat;
  cout  << pAnimal1->GetName() << " says "
    << pAnimal1->Speak() << endl;

  Animal *pAnimal2 = &cDog;
  cout  << pAnimal2->GetName() << " says "
    << pAnimal2->Speak() << endl;
  return 0;
}
```



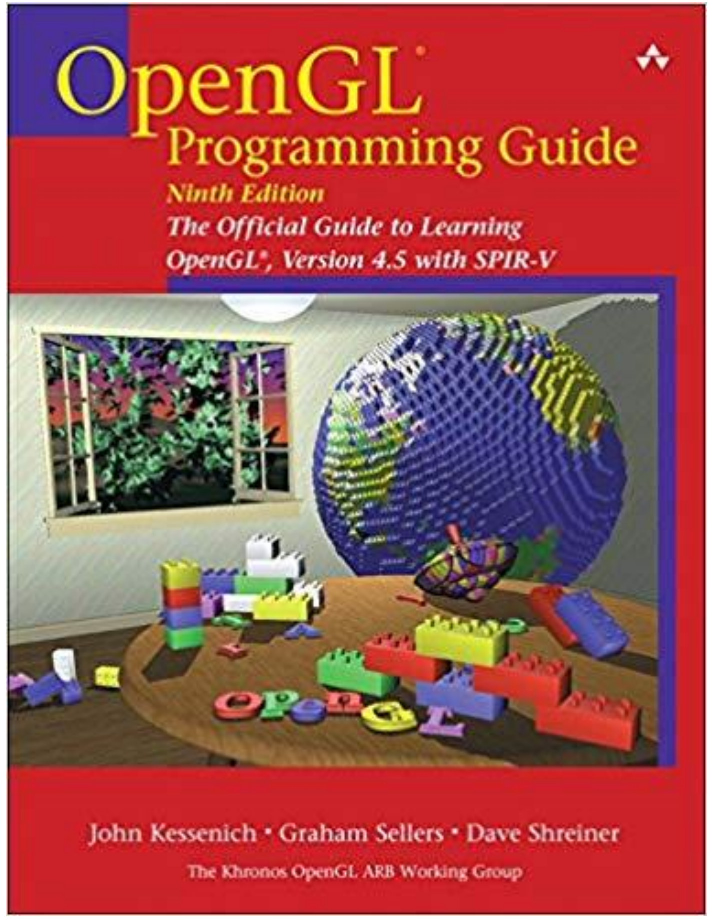
```
Fred says Meow
Garbo says Woof
Fred says ???
Garbo says ???
Press any key to continue . . .
```

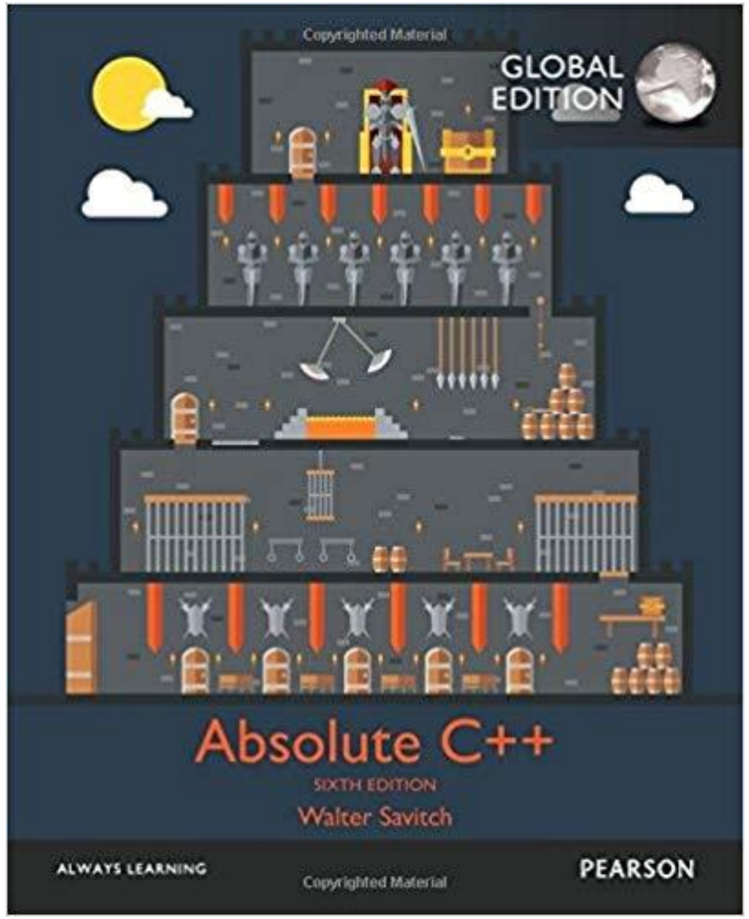
Thanks so far!

Any Questions?

Reading List

LearnOpenGL.com


OpenGL
Programming Guide
Ninth Edition
The Official Guide to Learning OpenGL®, Version 4.5 with SPIR-V
John Kessenich · Graham Sellers · Dave Shreiner
The Khronos OpenGL ARB Working Group



Some Exercises:

- https://coderbyte.com
- https://www.codewars.com/
- https://leetcode.com