Object Oriented Programming

You Know C, So C++ is Easy!

What we need to work with the modules?

- How will other programs interact with our module?
  - How to set up an interface
  - Which functions to make available to the user
  - Which data and functions are hidden from the user
  - How information is passed between various functions
- Application Programming Interface (API)
  - Specification required to make use of module
- Difference between module, library, and API?

What is an API

- Blackbox for software module, code library, subsystem, and even working components of an operating system or a remote server.
- Allows us to use a module, library, or service, without having to know the details of how it's implemented
- All we need to know is how to pass information to the components of API
- APIs in C
    - function declarations (in the .h file), along with any constants and other important values defined there
    - documentation (at the top of each function) that describes what each of the functions does and their parameters and return values, and
    - any documentation that is maintained externally ()

Examples of APIs

- Google Maps: Allows you to make use of the Google maps framework for plotting locations and for finding paths between points in maps - among many other things!
  - https://developers.google.com/maps/documentation/javascript/overview
- TensorFlow: Allows you to set up, train, test, evaluate, and operate a deep neural network for solving a task that requires learning from a very large dataset. Nowadays this is behind some of the most useful applications in A.I.
  - https://www.tensorflow.org
- Amazon AWS: Amazon's cloud-based AWS runs a large portion of internet-hosted services, and powers all kinds of applications from online trade to providing computing power for large simulations.
  - https://docs.aws.amazon.com/index.html#lang%2Fen_us
- Unity: Possibly the most popular API for creating, manipulating, and rendering 3D content, from graphical user interfaces and simulations, to interactive programs and games.
  - https://docs.unity3d.com/ScriptReference/

This week

- What we will cover:
  - Some missing concepts in C
  - Introduction to concepts in Object Oriented Programming OOP
- What we will not cover
  - C++ Syntax

Module - reason for “include guards”

guard prevents multiple replacements

basis.h

```
#ifndef_BASIS_H_
#define_BASIS_H_
//content of basis,
#define SIZE 1024 .. .
#endif
```

Interface file

1 preprocessing will replace "basis.h" with actual content of basis.h

stat.h

```
#include <stdio.h>
#include "basis.h"
```

graph.h

```
#include <stdio.h>
#include "basis.h"
```

depend on basis.h

```
#include <stdio.h>
#include "stat.h"      replaces basis.h
#include "graph.h"     replaces basis.h

int main
{
        //content of main.cpp
        return 0;
}
```

main.c

depend on stat.h, graph.h and basis.h

C APIs Examples flat C APIs ⇒ Pure API in C

- Standard C Library ⇒ written in C
  - Collection of include files (such as stdio.h, stdlib.h, and string.h) and library routines for I/O, string handling, memory management, mathematical operations, and so on
- The Windows API ⇒ written in C
  - **Win32 API**, this is the core set of interfaces used to develop applications for the Microsoft Windows range of operating systems
- The Linux Kernel API ⇒ written in C
  - The API includes driver functions, data types, basic C library functions, memory management operations, thread and process functions
- Image Libraries ⇒ written in C
  - Most of the open source image libraries are written in C. For example, the libtiff, libpng, libz, libungif, and jpeg libraries.

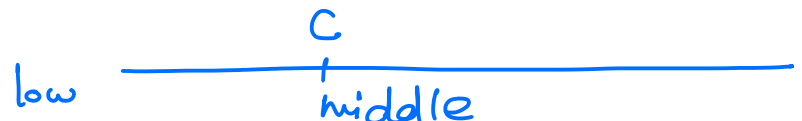
C features

The purpose of a programming language is to help express ideas in code.

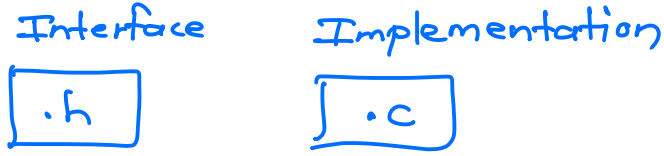
Two related tasks:

- Provide way to specify actions to be executed by the machine (requires a language that is close to the machine)
- Second, the programmer also requires a language which is "close to the problem to be solved", that the concepts of a solution can be expressed directly and concisely.

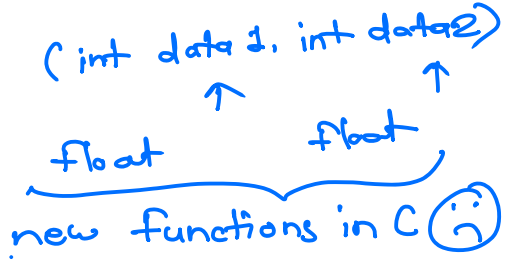
1. Built-in types such as `int`, `float`, `double`, `char`, and arrays and pointers to these.
2. Custom types created via the `typedef` and `enum` keywords.
3. Custom structures declared with the `struct` or `union` keywords.
4. Global free functions.
5. Preprocessor directives such as `#define`.



What's missing in C?



- C provides no way to hide or **protect sensitive data** from being misused or accessed in a way that was not intended by us when we developed the module.
- Difficult to **wrap data and functions for that data together**.
- C doesn't allow us to use **same function name with different parameter types, or different number of parameters,** leading to large number functions in API
- C doesn't have an elegant way of **extending** existing functions

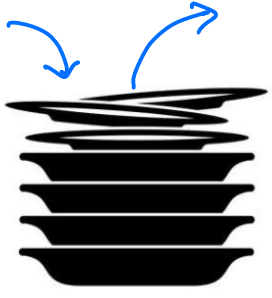


Abstract data type

The Stack ADT

characteristics

1. Linear data structure
2. Last In First Out (LIFO)



structure for the data + operations

seen in recursion

⇓

stack frame

The Stack ADT

Primary function

operations (user-defined)

- **Stack()** creates a new stack that is empty.
- **push(item)** adds a new item to the top of the stack.
- **pop()** removes the top item from the stack.
- **peek()** returns the top item from the stack but does not remove it.
- **isEmpty()** tests to see whether the stack is empty.
- **size()** returns the number of items on the stack.

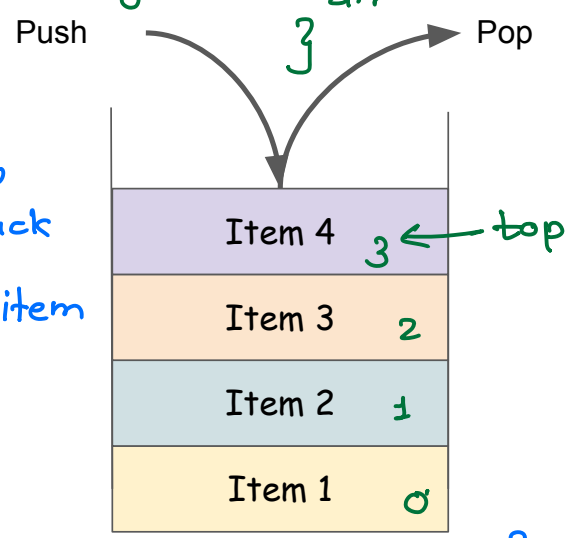
isFull() ⇒ limited capacity

helper functions

somehow keep track of top item

```
struct
{
  capacity ← 10
  top ← Index = 3
  * arr
}
```

pointer to the array



how do we implement it?

① array ② linked-lists

Stack Data Structure C

Interface (.h)

```
#ifndef STACK_H
#define STACK_H
/* A stack data structure */
struct stack
{
    int capacity;   // define max capacity of the stack
    int top;
    int *items;
};

/* Create a new stack data structure */
struct stack* StackCreate(int size);
/* Destroy an existing stack data structure */
void StackDestroy(struct stack* pt);
/* Push a new value onto the stack */
void StackPush(struct stack *pt, int x);
/* Pop the last value from the stack */
int StackPop(struct stack *pt);
/* Return 1 if the stack contains no values */
int isEmpty(struct stack *pt);
#endif
```

should move to implementation

dynamically create

you need to free the memory

Peek

Implementation (.c)

```
/* Definitions of all the functions */
// Utility function to initialize the stack
struct stack* StackCreate(int size){
    struct stack *pt = (struct stack*)malloc(sizeof(struct stack));

    pt->capacity = size;
    pt->top = -1;
    pt->items = (int*)malloc(sizeof(int) * size);

    return pt;
}
```

Stack Data Structure C using opaque data type

(for your knowledge)
not covered in exam

Interface

```
##ifndef STACK_H
#define STACK_H
/* An opaque stack data structure type */
struct stack;

/* Create a new stack data structure */
struct stack* StackCreate(int size);
/* Destroy an existing stack data structure */
void StackDestroy(struct stack* pt);
/* Push a new value onto the stack */
void StackPush(struct stack *pt, int x);
/* Pop the last value from the stack */
int StackPop(struct stack *pt);
/* Return 1 if the stack contains no values */
int isEmpty(struct stack *pt);

#endif
```

← declaration

Implementation

← definition

```
struct stack
{
    int capacity;    // define max capacity of the stack
    int top;
    int *items;
};
/* Create a new stack data structure */
struct stack* StackCreate(int size){
    struct stack *pt = (struct stack*)malloc(sizeof(struct stack));

    pt->capacity = size;
    pt->top = -1;
    pt->items = (int*)malloc(sizeof(int) * size);

    return pt;
}

/* Definitions of all the remaining functions */
```

The problem we solved using opaque data type

- Need of an opaque data type to create stack structure
- What is an opaque data type??
  - Way to achieve data hiding in C
  - **Hide the implementation details of an interface from ordinary clients**, so that the implementation may be changed without the need to recompile the modules using it.
  - Read more from source: https://en.wikipedia.org/wiki/Opaque_data_type
  - Data definitions of CDT are not available to the user because of opaque data type
- Benefits?
  - Simplifies the API. The user does not need to know or care about internal data.
  - The struct definition can change without impacting any of the using code.
  - Prevents misuse of internal struct data.

Opaque data type you have seen before ⇒ used in assignments and exercises

you get a file pointer

```
FILE * fp = fopen (filename, "w");

fwrite( .., .., …, fp);

fprintf(fp, "Hello Word");

fclose(fp);
```

what is fp? How is it organized?

⇒ we don't know that's okay

fp ⇒ opaque data pointer

Any other issues with the code in C?

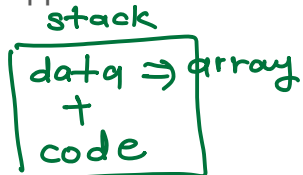
Interface

```
##ifndef STACK_H
#define STACK_H
/* An opaque stack data structure type */
struct stack;

/* Create a new stack data structure */
struct stack* StackCreate(int size);
/* Destroy an existing stack data structure */
void StackDestroy(struct stack* pt);
/* Push a new value onto the stack */
void StackPush(struct stack *pt, int x);
/* Pop the last value from the stack */
int StackPop(struct stack *pt);
/* Return 1 if the stack contains no values */
int isEmpty(struct stack *pt);
#endif
```

Implementation

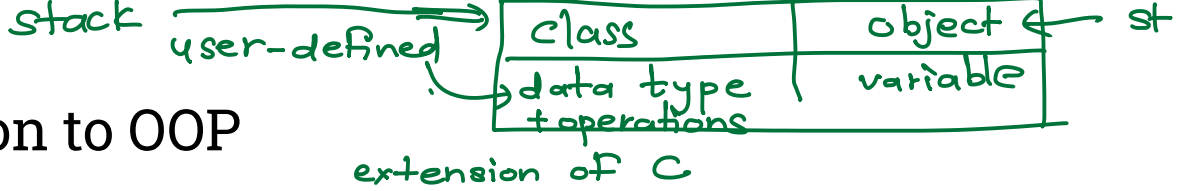
```
struct stack
{
    int capacity;    // define max capacity of the stack
    int top;
    int *items;
};
/* Definitions of all the functions */
```

- Organization of code could be improved
- Data and code that operate on the data could be grouped together
- Create an individual, self contained unit for each stack
- This unit should also control access to data managed by module to support information hiding



Allez-OOP!

- Involves the use of **objects** (instances of classes) as the central theme
  - Encapsulation - Grouping together data with the code that processes it, and treating it as a unit
- Many OOP languages: Java, C++, Python
- Benefits of OOP
  - Secure, protects information through encapsulation
  - OOP models complex things as reproducible, simple structures
  - Reusable, OOP objects can be used across programs
  - Allows for class-specific behavior through polymorphism
  - Easier to debug, classes often contain all applicable information to them

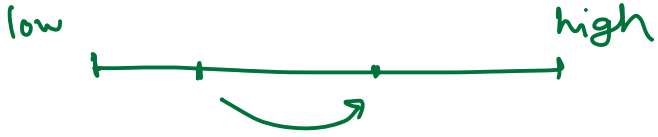


Introduction to OOP

- Object Oriented Programming is a model for developing software components that is based on the idea of **encapsulation**.
- C++ was known by the name "**C with classes**" up until about 1985
- Other features OOP offers:
  - **Abstraction** as the process of refining away the unimportant details of an object
  - **Class** as user defined type like CDT, or int, bool
  - **Object** as the fundamental unit of information storage, processing, and manipulation int a, bool b
  - **Encapsulation** as grouping together the types, data, and functions that make up a class.
  - **Inheritance** allowing one class to receive the data structures and functions described in a simpler base class
  - **Polymorphism** ability of one type to appear as and to be used like another type. Possible due to dynamic binding

object taking diff. forms

OOP languages



- Many languages: Python, Java, C++
- In 1983, the first vestiges of C++ were created by Bjarne Stroustrup.
- Motivation: to make C
  - Fast
  - Simple to Use
  - Cross-platform
  - High level features

Abstraction - Extracting Out the Essential Characteristics of a Thing

- Abstraction in useful in software
  - hide irrelevant detail, and concentrate on essentials.
  - present a "black box" interface to the outside world. The interface specifies the valid
  - operations on the object, but does not indicate how the object will implement them internally.
  - break a complicated system down into independent components. This in turn localizes
  - knowledge, and prevents undisciplined interaction between components.
  - reuse and share code.
- C support abstraction?
  - How?
    - Through allowing the user to define new types (struct, enum) that are almost as convenient as the predefined types (int, char, etc.), and to use them in a similar way.
    - Header files
- C++ support abstraction?
  - Through **Classes, Header files, access modifiers** (1) (2) (3) they work altogether to provide abstraction + Encapsulation

Stack Data Structure C++

Interface

```
#ifndef STACK_H
#define STACK_H
class Stack
{
    Encapsulate Stack Data and
    functions that operate on Stack
};
#endif
```

- Encapsulation: Bundle together data and code that operates on that data

Encapsulation - Grouping Together Related Types, Data, and Functions

- When you bundle together an abstract data type with its operations, it is termed "**encapsulation**".
- In C:
  - There is no way to tell a C compiler,
    - "These three functions are the only valid operations on this particular struct type."
  - There is no way to prevent a programmer from defining additional functions that access the struct in an unchecked or inconsistent manner.
- The C++ class mechanism provides **OOP encapsulation**.
  - A class is the software realization of encapsulation.
  - A class is a type, just like `char, int, double,` and `struct rec *` are types, and so you must declare variables of the class to do anything useful.
  - You can do pretty much anything to a **class** that you can do to a type, such as take its size, or declare variables of it.
  - You can pretty much do anything to an **object** (instance of a class) that you can do to a variable

Encapsulation in OOP

```
#ifndef STACK_H
#define STACK_H

class Stack
{
    Private:
        int *arr;
        int top;
        int capacity;

    Public:
        Stack();        // constructor
        ~Stack();       // destructor

        void StackPush(int);
        int StackPop();
        int StackPeek();

        int StackIsEmpty();
        int Stack_length();

};
 #endif
```

access modifiers

data

functions

only these functions give access to private data

Non OOP

.h

```
#ifndef STACK_H
#define STACK_H
/* An opaque pointer to a stack data structure */
typedef struct Stack *StackPtr;

/* Create a new stack data structure */
StackPtr StackCreate();
/* Destroy an existing stack data structure */
void StackDestroy(StackPtr stack);
/* Push a new value onto the stack */
void StackPush(StackPtr stack, int val);
/* Pop the last value from the stack */
int StackPop(StackPtr stack);
/* Return 1 if the stack contains no values */
int StackIsEmpty(const StackPtr stack);
#endif
```

functions

hide the data

.c

```
struct Stack
{
    int top;
    int size;
    int* array;
};
StackPtr StackCreate()
{
    return (StackPtr) calloc(sizeof(struct Stack), 1);
}
/* Definitions of all the functions */
```

data

Information Hiding using Classes

when object is created for a class, only then memory is allocated

```
#ifndef STACK_H
#define STACK_H

class Stack
{
private:
    int *arr;
    int top;
    int size;

public:
    Stack();                // constructor
    ~Stack();                 // destructor

    void StackPush(int);
    int StackPop();
    int StackPeek();

    int StackIsEmpty();
    int Stack_length();
};
#endif
```

- A class is just a user-defined type with all the operations on it.
- A class is often implemented as a **struct of data,** grouped together with **functions that operate on that data**.
- The compiler imposes strong typing—ensuring that these functions are only invoked for objects of the class, and that no other functions are invoked for the objects.

Access Modifiers

```
#ifndef STACK_H
#define STACK_H
// Define the default capacity of the stack

class Stack
{
      private:
          int *arr;
          int size;
          void update_length();

      public:
          Stack();                  // constructor
          ~Stack();                 // destructor

          void StackPush(int);
          int StackPop();
          int StackPeek();

          int StackIsEmpty();
          int Stack_length();
};
```

3 important access modifiers

- Public:
  - The declarations are visible outside the class and can be set, called, and manipulated as desired.
- Private:
  - The declarations can only be used by the member functions of this class. Private declarations are visible outside the class (the name is known), but they are not accessible.
- Protected:
  - Visible to functions inside this class, and to functions in classes derived from this class

Inheritance

Constructor

- Automatically called by the compiler when **new instance** of that class is declared
- A constructor has the same name as the class and no return value
- Task of a constructor:
  - It sets the initial values of the object
  - ensures that objects will always have valid data to work on.
- Declaration
  - Constructors can be identified by their names. In contrast to other member functions,
  - the following applies:
    - the name of the constructor is also the class name
    - a constructor does not possess a return type—**not even void.**
  - Constructors are normally declared in the public section of a class.
- Definition:
  - `Class_name::Class_name`

object

⇒ assigns default values

cannot have return type as void

function_name()

without constructor, your program may / may not work
so it does bare min work to make your prog. work.
Cannot rely on it !!

Constructor

```
#include <iostream>
#include "stack.h"

// Constructor to initialize the stack
Stack::Stack()
{
          arr = new int[20];
          capacity = 20;
          top = -1;
};
```

no return type

- A constructor without parameters is referred to as a **default constructor**.
- The default constructor is only called if an object definition does not explicitly initialize the object.
- A default constructor will use standard values for all data members.

Destructor

- Task of destructor:
  - Releasing memory and closing files
- Declaration:
  - Declared in public section:
    - `~class_name();`

tilde ~

- Calling destructors:
  - A destructor is called automatically at the end of an object's lifetime:

No need to call it.

when program exits, it calls the destructor

Program doesn't compile without destructor

Destructor

```
#include <iostream>
#include "stack.h"

// Destructor to free memory allocated to the stack
Stack::~Stack() {
    delete[] arr;
    std::cout << "stack is destroyed!!" << std::endl;
}
```

Destructor to free memory allocated to the stack

Object

- Instance (or a variable) of class
- Class provides a template for building objects of that class to use in our programs
- Defined in the client program

client.cpp

```
1.  int main()
2.  {
3.      // create a new stack
4.      Stack st;
5.      //Push int on top of stack
6.      st.StackPush(1);
7.      st.StackPush(4);
8.      //pop an element
9.      st.StackPop();
10.     cout << "Top of the stack is: " << st.top << endl;
11.
12.     return 0;
13. }
```

calls constructor

class stack {
private: int *arr;
int capacity;
int top;
public: push
pop
peek
}

use st.peek() instead

will it work? NO!! cannot access st.top ; its

Current Stack Module

in CPP

private

stack st(10); //creates 10 elements

Stack.h

```
#ifndef STACK_H
#define STACK_H

class Stack
{
private:
    int *arr;
    int top;
    int size;

public:
    Stack();
    ~Stack();

    void StackPush(int);
    int StackPop();
    int StackPeek();

    int StackIsEmpty();
    int Stack_length();
};
#endif
```

Stack.cpp

```
#include <iostream>
#include "stack.h"

// Constructor to initialize the stack
Stack::Stack()
{
      arr = new int[20];
      capacity = 20;
      top = -1;

};
// Destructor to free memory allocated
to the stack
Stack::~Stack() {
    delete[] arr;
}
…
```

main.cpp

```
int main()
{
    // create a new stack
    Stack st;
    //Push int on top of stack
    st.StackPush(1);
    st.StackPush(4);
    //pop an element
    st.StackPop();
    return 0;
}
```

no need to call destructor

Stack with added functionality in C

- We want to give an option to our client to add (push) an **array of elements** onto our stack.
- In C,
  - We will need to add another function with different name to accommodate different set of parameters

```
// Utility function to add an element `x` to the stack
void StackPush(struct stack *pt, int x);
// Utility function to add array of elements on stack
void StackPush_array(struct stack *pt, int x[], int size);
```

size of the array we want to push

- Requires all our modules using our API to be modified and recompiled
- Also, what if there is a bug in functionality of **Push** operation?
  - Need to update both `StackPush` and `StackPush_array`

methods are functions declared in the class

Stack with added functionality in C++

- We want to give an option to our client to add (push) an **array of elements** onto our stack.
- In C++
  - **Method overloading:** We can use the same function name with different name to accommodate different set of parameters

```
void StackPush(int);
void StackPush(int[], int);
```

array

size

- Now, the class has two methods with the same name
- The compiler will select which one to call based on the user's code (by looking at the types of parameters)

similar to operator overloading

a = 5, b = 10, c = 0, c = a + b

a = [5], b = [10], c = [ ] = c

[15]

Method Overloading with constructors

- Overloaded constructors have the **same name** and **differ by number** and **type of arguments**.
- While compiling, a constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.
- For Stacks,
  - Default Constructor:
    - `Stack()`
  - Non-Default Constructor:
    - `Stack(int size)`
  - Copy Constructor:
    - `Stack(const Stack &in_stack);`
  - Assignment Operator
    - `Stack &operator = (const Stack &in_stack);`

st2

Stack st (&st2)

st

address of another stack

Stack st = st2;

st already exists

Method Overloading with constructors

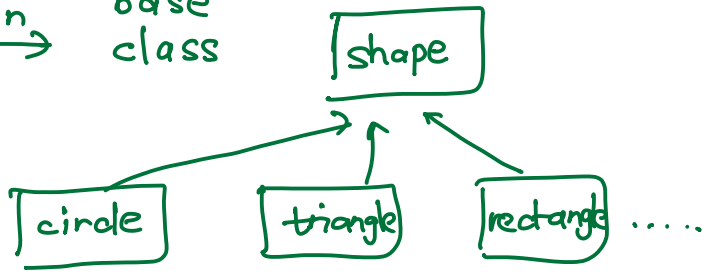
```
// Default Constructor to initialize the stack
Stack::Stack()
{
    arr = new int[20];
    capacity = 20;
    top = -1;
}
```

```
// non-default constructor
Stack::Stack(int size)
{
    arr = new int[size];
    capacity = size;
    top = -1;
}
```

```
// copy constructor
Stack::Stack(const Stack &in_stack)
{
    capacity = in_stack.capacity;
    arr = new int[capacity];
    top = capacity - 1;
    std::copy(in_stack.arr , in_stack.arr+capacity,
arr);
}
```

```
//Assignment operator
Stack &Stack::operator = (const Stack &in_stack)
{
    if(this!=&in_stack)
    {
        delete[] arr;
        capacity = in_stack.capacity;
        arr = new int[in_stack.capacity];
        std::copy(in_stack.arr , in_stack.arr+capacity, arr);
    }
    return *this;
}
```

shared common properties → base class



derived class → class

derived class are publicly available & maybe hide common properties

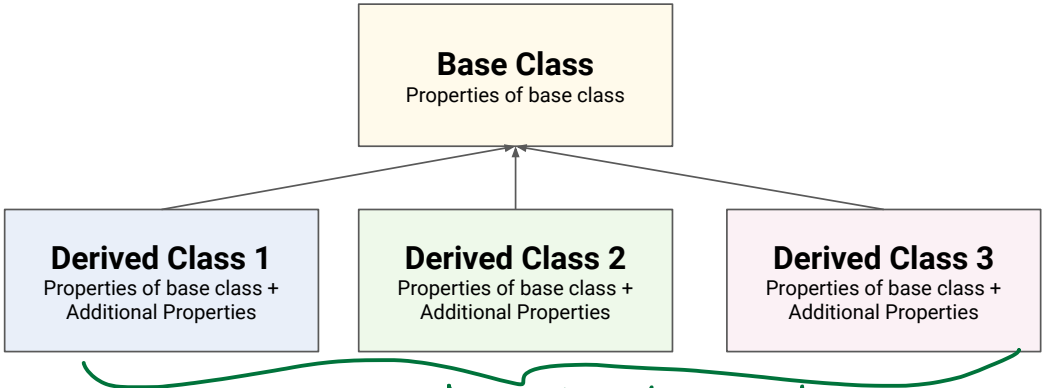
Inheritance

⇒ Reuse of code

⇒ Provide abstraction

Inheritance

- Ability of an object-oriented language to build and use a hierarchy of classes representing entities that are related to each other and share some characteristics.
- Inheritance allows new classes to be constructed on the basis of existing classes.
- The new **derived class** “inherits” the data and methods of the so-called **base class**.
- But you can add more characteristics and functionality to the new class.

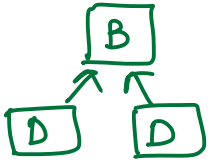


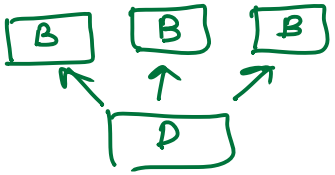
do not share their own properties with each other

Inheritance benefits

- Data abstraction:
  - General characteristics and abilities can be handled by generic (base) classes
  - Specializations can be organized in hierarchical relationships by means of derived classes.
- Re-usability:
  - Classes that you have defined and tested can be reused and adapted to perform new tasks.
  - The base class implementation need not be known for this purpose

Base class vs Derived class





multiple inheritance

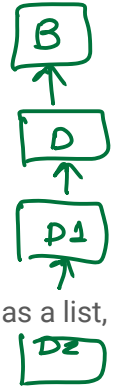
Is-A, Has-A, As-A Relationships Among Classes

Hypothetical classes `Class1` and `Class2`

think of a logical relationship between them that can be written:

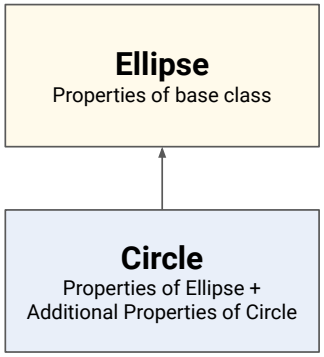
- If writing "`Class1 is-a Class2`" is best, for example,
  - A savings account is an account
  - then Class1 should be a derived class (a subclass) of Class2.
- If writing "`Class1 has a Class2`" is best, for example
  - A cylinder has a circle as its base
  - then Class1 should have a member variable of type Class2.
- In the case of `Class1 is implemented as a Class2`, as in the stack is implemented as a list,
  - then Class1 should be derived from Class2, but with private inheritance.
  - This is by far the least common case!

most common



pitfalls of Inheritance

Inheritance - Example of bad design

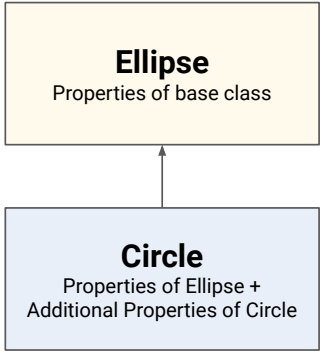



```
Class Ellipse{
  public:
        Ellipse();
        Ellipse(float major, float minor);

        void SetMajorRadius(float major);
        void SetMinorRadius(float minor);
        float GetMajorRadius() const;
        float GetMinorRadius() const;
  private:
        float mMajor;
        float mMinor;
};
```

```
Class Circle: public Ellipse{
public:
        Circle();
        explicitCircle(float r);

        void SetRadius(float r);
        float GetRadius() const;
};

void Circle::SetRadius(float r) {
        SetMajorRadius(r);
        SetMinorRadius(r);
}

float Circle::GetRadius() const {
        return GetMajorRadius();
}
```

- Need to follow certain rules with Inheritance:
- **Liskov Substitution Principle**:
  - If S is a subclass of T then objects of type T can be replaced by objects of type S without any change in behaviour.
  - It should always be possible to substitute the base class for a derived class without any change in its behaviour

Inheritance - Example of bad design

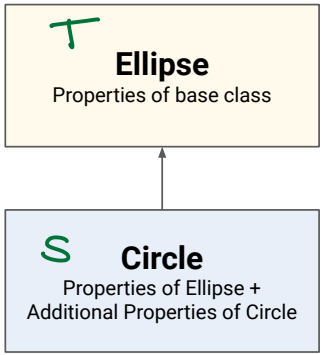



```
Class Ellipse{
  public:
        Ellipse();
        Ellipse(float major, float minor);

        void SetMajorRadius(float major);
        void SetMinorRadius(float minor);
        float GetMajorRadius() const;
        float GetMinorRadius() const;
  private:
        float mMajor;
        float mMinor;
};
```

```
Class Circle::public Ellipse{
public:
        Circle();
        explicitCircle(float r);

        void SetRadius(float r);
        float GetRadius() const;
};

void Circle::SetRadius(float r) {
        SetMajorRadius(r);
        SetMinorRadius(r);
}

float Circle::GetRadius() const {
        return GetMajorRadius();
}
```

- Reason this is a bad design:
  - Circle will also inherit and expose the **`SetMajorRadius()`** and **`SetMinorRadius()`** methods of Ellipse.
  - Object of class Circle can be passed to code that accepts Ellipse
  - These could be used to break the self-consistency of our circle by letting users change one radius without also changing the other.

Inheritance - Example of bad design




```
Class Ellipse{
  public:
        Ellipse();
        Ellipse(float major, float minor);

        void SetMajorRadius(float major);
        void SetMinorRadius(float minor);
        float GetMajorRadius() const;
        float GetMinorRadius() const;
  private:
        float mMajor;
        float mMinor;
};
```

```
Class Circle::public Ellipse{
public:
        Circle();
        explicitCircle(float r);

        void SetRadius(float r);
        float GetRadius() const;
};

void Circle::SetRadius(float r) {
  ..
}

float Circle::GetRadius() const {
        ..
}
```

- How does it break Liskov Substitution Principle?
- If S is a subclass of T then objects of type T can be replaced by objects of type S without any change in behaviour.
- Cannot replace uses of Ellipse with Circle without breaking behavior

```
void TestEllipse(Ellipse &e) {
   e.SetMajorRadius(10.0);
   e.SetMinorRadius(20.0);
   assert(e.GetMajorRadius()==
10.0 && e.GetMinorRadius()==
20.0);
}
...
Ellipse e;
Circle c;
TestEllipse(e);
TestEllipse(c);// fails!
```

Public, Private and Protected Inheritance

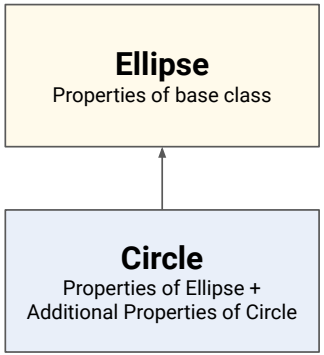
What is Accessible Where?

- **Public** inheritance makes public members of the base class public in the derived class, and the protected members of the base class remain protected in the derived class.
- **Protected** inheritance makes the public and protected members of the base class protected in the derived class.
- **Private** inheritance makes the public and protected members of the base class private in the derived class.

C++ defaults to Private inheritance

Inheritance - Example of good design

object of type circle doesn't have access to all the public functions of Ellipse




```
Class Ellipse{
  public:
        Ellipse();
        Ellipse(float major, float minor);

        void SetMajorRadius(float major);
        void SetMinorRadius(float minor);
        float GetMajorRadius() const;
        float GetMinorRadius() const;
  private:
        float mMajor;
        float mMinor;
};
```

```
Class Circle: private Ellipse{
public:
        Circle();
        explicitCircle(float r);

        void SetRadius(float r);
        float GetRadius() const;
};

void Circle::SetRadius(float r) {
        SetMajorRadius(r);
        SetMinorRadius(r);
}

float Circle::GetRadius() const {
        return GetMajorRadius();
}
```

all they can access is SetRadius() and GetRadius

- Private Inheritance:
  - Let's you inherit the functionality, but not the public interface of another class.
  - All public members of base class (Ellipse) become private members of derived class.
  - Circle doesn't expose any SetMajorRadius and SetMinorRadius methods of Ellipse.

Another Inheritance Example(Stack Inheriting from List)


```
template <class T>
class stack : private std::list<T>
{
      public:
            stack() {}
            stack(stack<T> const& other) : std::list<T>( other ) {}
            ~stack() {}
            void push( T const& value ) { this->push_back( value ); }
            void pop() { this->pop_back(); }
            T const& top() const { return this->back(); }
            int size() { return std::list<T>::size(); }
            bool empty() { return std::list<T>::empty(); }
};
```

user of stack interface cannot access list<T> functions

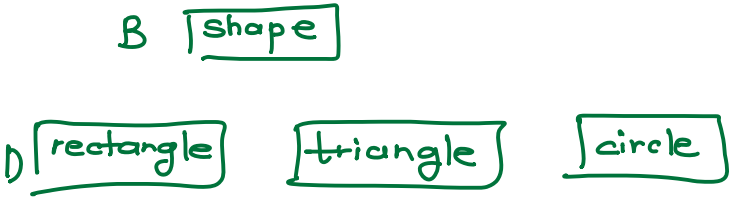
- Private inheritance **hides the list member functions** from the outside world.
- Member functions are **still available to** the member functions of the **stack<T> class**
- When same member function defined in base class (list) is used in derived class, the name of the base class is followed by ::

https://en.cppreference.com/w/cpp/container
https://en.cppreference.com/w/cpp/container/list

Inheritance considerations for API design

- Only use inheritance where appropriate.
  - Deciding whether a class should inherit from another class is one of the most difficult part of software design.
- Avoid deep inheritance trees
  - Deep inheritance hierarchies increase complexity and invariably result in designs that are difficult to understand and software that is more prone to failure.
- Don't overdesign
  - If you have a base class that is inherited by only a single class in your entire API, this is an indication that you have over-designed the solution for the current needs of the system
- SOLID Principle: (acronym for the first five object-oriented design (OOD) principles by Robert C Martin)https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
  - S - Single-responsibility Principle
  - O - Open-closed Principle
  - **L - Liskov Substitution Principle**
  - I - Interface Segregation Principle
  - D - Dependency Inversion Principle

no need to know

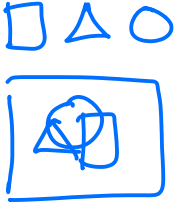


object of type 'shape' can take diff. forms : It can be rectangle, triangle, circle

Inheritance + Polymorphism is very powerful!

Polymorphism

Extension to inheritance

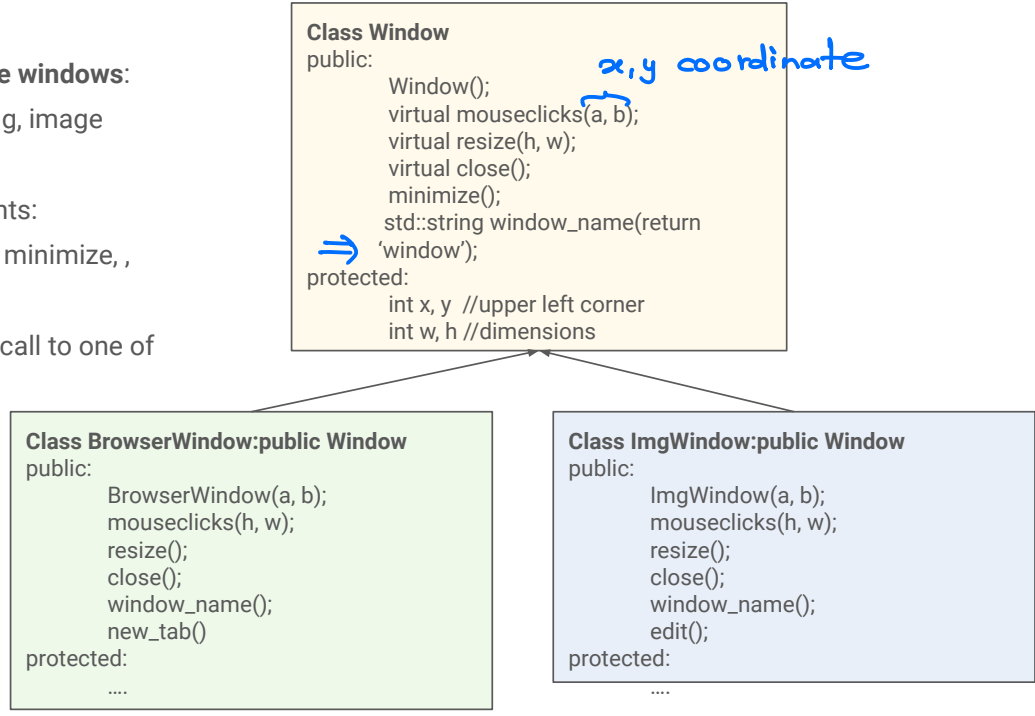


Ability for objects to take many different forms

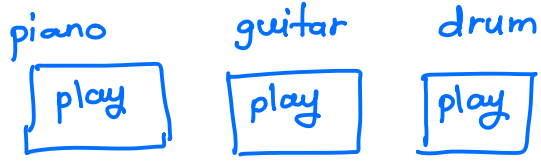
Window Manager of your machine

- The window manager has a list of **active windows**:
  - Browser window, word processing, image window, chat window
- Each window responds to series of events:
  - mouse clicks, refresh, maximize, minimize, , close window, shutdown
  - Each event generates a function call to one of the **objects** on window list

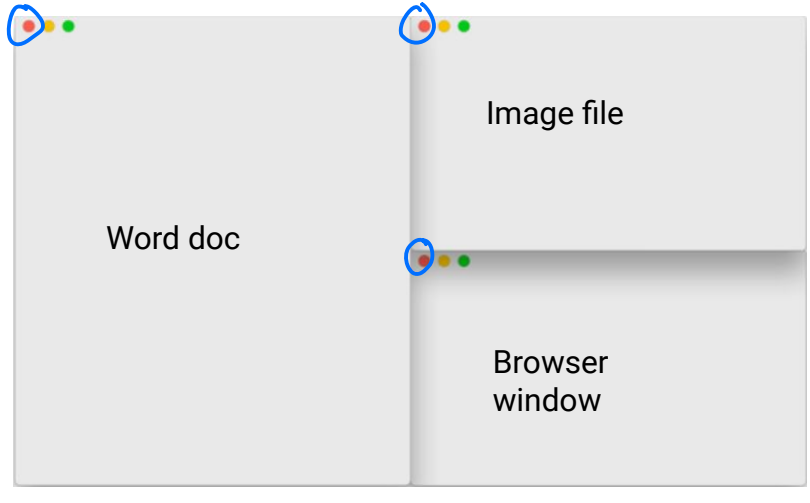
Common functions apply to all window objects



Window Manager



- What happens when you close a word processing file?
  - Before closing, you are asked whether you want to save the file?
  - Window is closed
- What happens when you close the Image file?
  - Window is closed
- What happens when you close the browser window?
  - Depends!
  - If multiple open tabs, depending on your browser and OS, all the open tabs info is saved
  - Close the window
- **Same function** - closing the window
- **Different action** depending on which window to close



you want to create a shortcut key to close all the windows

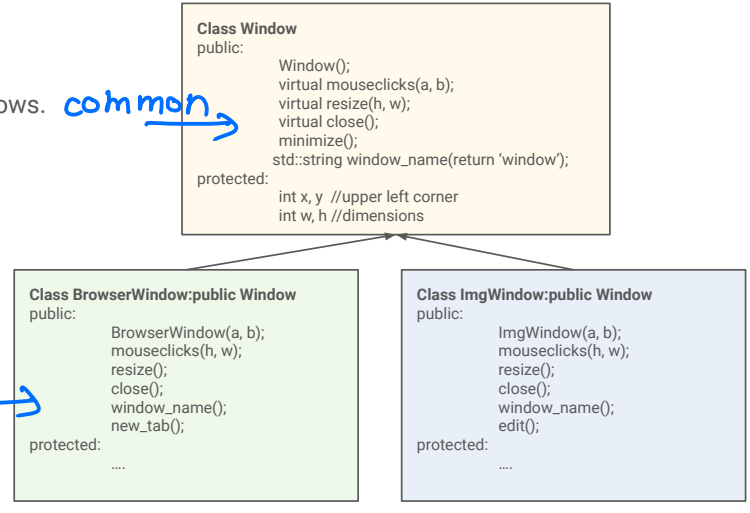
Window Manager Task - Close all windows at once

- Write individual code to close the window for each different windows.
- Some window functions will be common for all windows:
  - Minimizing the window
  - Enter full screen
  - Various mouse clicks
- Some window functions are different:
  - Closing the window
  - Typing in the window
  - ...
- Non OOP approach: Write all individual functions for all windows
- OOP approach:
  - take advantage of inheritance and create hierarchical representation of related classes and where **common data and methods are implemented by a parent class**, and **specific behaviour is implemented by derived (children) classes**
  - **Create objects of class window that can take many forms**

common

specific

base class



Window Manager of your machine

- What is a virtual function?
- member function which is declared within a base class and is re-defined by a derived classes

Class Window
public:
Window();
virtual mouseclicks(a, b);
virtual resize(h, w);
virtual close();
minimize();
std::string window_name(return 'window');
protected:
int x, y //upper left corner
int w, h //dimensions

declare the function

Class BrowserWindow:public Window
public:
BrowserWindow(a, b);
mouseclicks(h, w);
resize();
close();
window_name();
new_tab();
prf switch_profile(prf p, prf p2);
protected:
struct prf;
....

Class ImgWindow:public Window
public:
ImgWindow(a, b);
mouseclicks(h, w);
resize();
close();
window_name();
edit();
protected:
....

BrowserWindow functions

```
BrowserWindow *bw = new BrowserWindow(4, 250);
std::cout << "Win name = " << bw->window_name();
bw->new_tab();
bw->new_tab();
Profile = bw->switch_profile(p, p2);
bw->close();
//close the browser by saving the data for open tabs
```

2 tabs

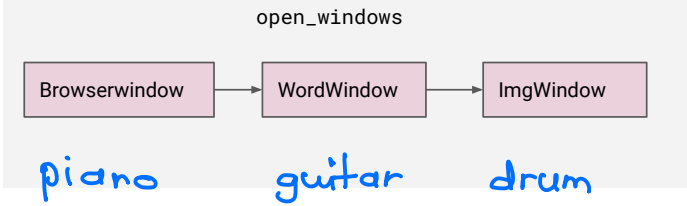
List of Windows

- List of various windows:
  - `std::list<Window*> open_windows;`
- Create various windows and add them to list of open_windows
  - `Window * win = new BrowserWindow;`
  - `open_windows.push_back(win);`
  - `win = new WordWindow;`
  - `open_windows.push_back(win);`
  - `win = new ImgWindow;`
  - `open_windows.push_back(win);`

List of windows named open_windows

win is of type Window

object of type window can take many diff forms



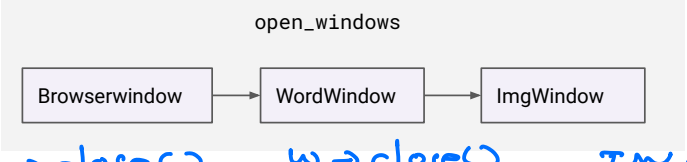
Class Window
public:
Window();
virtual mouseclicks(a, b);
virtual resize(h, w);
virtual refresh();
minimize();
std::string window_name();
protected:
int x, y //upper left corner
int w, h //dimensions

Class BrowserWindow:public Window
public:
BrowserWindow(a, b);
mouseclicks(h, w);
resize();
close();
window_name();
new_tab();
prf switch_profile(prf p, prf p2);
protected:
struct prf;
....

Class ImgWindow:public Window
public:
ImgWindow(a, b);
mouseclicks(h, w);
resize();
refresh();
window_name();
edit();
protected:
....

Close all the open windows

Traverse the open_windows list and ~~minimize~~ close each one of the object




```
//traverse the linked list of various different objects
for ( std::list::iterator p = open_windows.begin(); p != open_windows.end(); ++p )
{
      if ( (*p) is not closed )
      {
         (*p)->close(); //perform window specific operations to close the window
      }
}
```

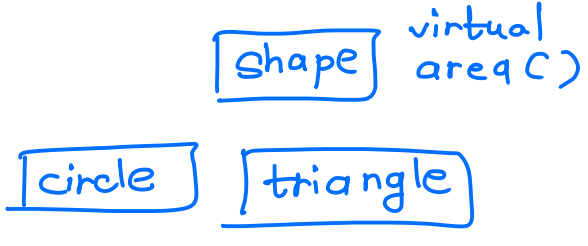
Polymorphism in API

- When building software systems, polymorphism provides the ability for objects to take many different forms, typically relying on inheritance.
- Objects of different types to be used interchangeably as long as they conform to the same interface.
- This is possible because the C++ compiler can delay checking the type of an object until run time, a technique known as late or dynamic binding.

Abstract Classes

only declared in base class
⇩ and defined in the derived class

- An abstract class is a class that has at least one **pure virtual function** (i.e., a function that has no definition).
- The classes inheriting the abstract class must provide a definition for the pure virtual function; otherwise, the subclass would become an abstract class itself.
- Essential to providing an abstraction to the code to make it reusable and extendable



Summary of A48

- Programming in C
- Abstract Data Types and Data Structures
- Multiple common data structures
- Computational complexity
- Tree Structure
- Graphs
- Recursion
- API design
- OOP principles

Thank you for wonderful Semester
All the best for finals!
Hope to see you in other classes
:) :)