CS601: Software Development for Scientific Computing

Autumn 2021

Week4:

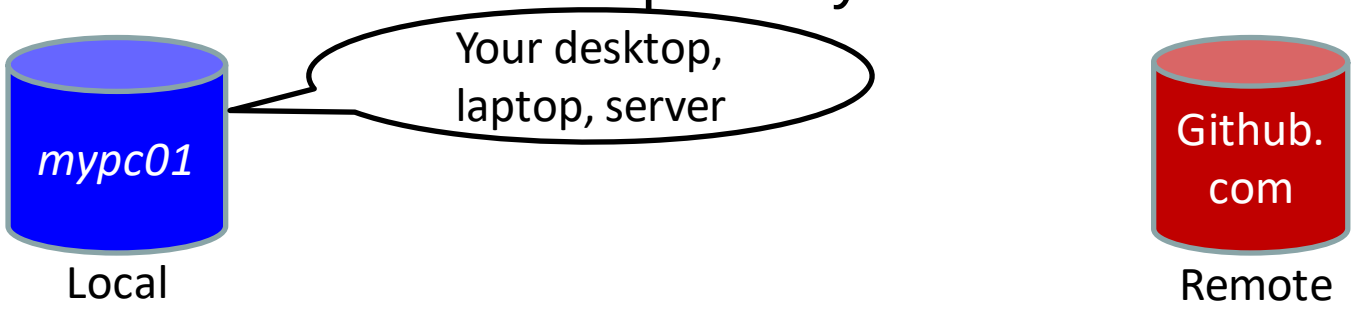
- Tools - Version Control System (Git and GitHub), Build tool (GNU make), GNU Debugger (gdb)
- Intermediate C++ (OO concepts)
- Structured Grids (contd..)

Last Week..

- Intermediate C++
  - Preprocessor directives, streams, and namespaces
- Structured Grids
  - PDEs and categories, the mathematical model, approximation, algebraic equations. Case study: 1D heat equation.
  - Program Representation???

Git

- Example of a Version Control System
  - Manage versions of your code – access to different versions when needed
  - Lets you collaborate
- 'Repository' – term used to represent storage
  - *Local* and *Remote* Repository



Git – Creating Repositories

- Two methods:
  1. ‘Clone’ / Download an *existing* repository from GitHub

Git – Creating Repositories

- Two methods:

  2. Create local repository first and then make it available on GitHub

Method 1: `git clone` for creating local working copy

- 'Clone' / Download an existing repository from GitHub – get your own copy of source code
  - `git clone` (when a remote repository on GitHub.com exists)

```
nikhilh@ndhpc01:~$ git clone git@github.com:IITDhCSE/dem0.git
Cloning into 'dem0'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
nikhilh@ndhpc01:~$
```

Method 2: `git init` for initializing local repository

- Create local repository first and then make it available on GitHub

  1. `git init`

  converts a directory to Git local repo

```
nikhilh@ndhpc01:~$ mkdir dem0
nikhilh@ndhpc01:~$ cd dem0/
nikhilh@ndhpc01:~/dem0$ git init
Initialized empty Git repository in /home/nikhilh/dem0/.git/
nikhilh@ndhpc01:~/dem0$ ls -a
.  ..  .git
```

`git add` for staging files

2. `git add`

'stage' a file i.e. prepare for saving the file on local repository

```
nikhilh@ndhpc01:~$ ls -a dem0/
.  ..  README
nikhilh@ndhpc01:~$ cd dem0/
nikhilh@ndhpc01:~/dem0$ git init
Initialized empty Git repository in /home/nikhilh/dem0/.git/
nikhilh@ndhpc01:~/dem0$ git add README
```

Note that creating a file, say, README2 in dem0 directory does not *automatically* make it part of the local repository

`git commit` for saving changes in local repository

3. `git commit`

'commit' changes i.e. save all the changes (adding a new file in this example) in the local repository

```
nikhilh@ndhpc01:~/dem0$ git commit -m "Saving the README file in local repo."
[master (root-commit) 99d0a63] Saving the README file in local repo.
 1 file changed, 1 insertion(+)
 create mode 100644 README
```

How to save changes done when you must overwrite an existing file?

Method 2 only: `git branch` for branch management

4. `git branch –M master`

rename the current as '`master`' (-M for force rename even if a branch by that name already exists)

```
nikhilh@ndhpc01:~/dem0$ git branch -M master
```

Method 2 only: `git remote add`

5. `git remote add origin git@github.com:IITDhCSE/dem0.git` – prepare the local repository to be managed as a tracked repository

```
nikhilh@ndhpc01:~/dem0$ git remote add origin git@github.com:IITDhCSE/dem0.git
```

command to manage remote repo.

associates a name 'origin' with the remote repo's URL

The URL of the repository on GitHub.com.

- This URL can be that of any other user's or server's address.
- uses SSH protocol
  - HTTP protocol is an alternative. Looks like: https://github.com/IITDhCSE/dem0.git

Method 2 only: GitHub Repository Creation

5.a) Create an empty repository on GitHub.com

(name must be same as the one mentioned previously – `dem0`)

Remote

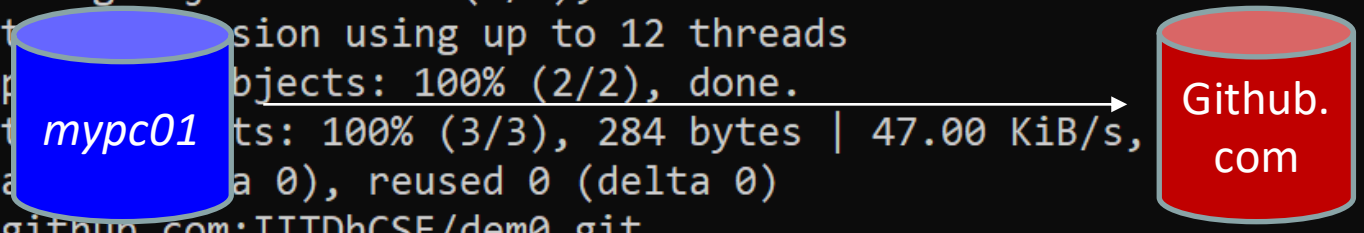
`git push` for saving changes in remote repo

`6. git push –u origin master`

'push' or save all the changes done to the 'master' branch in local repo to remote repo. *(necessary for guarding against deletes to local repository)*

`syntax: git push <remotename> <branchname>`

```
nikhilh@ndhpc01:~/dem0$ git push -u origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delt          sion using up to 12 threads
Comp          bjects: 100% (2/2), done.
Writ          ts: 100% (3/3), 284 bytes | 47.00 KiB/s,
Tota          a 0), reused 0 (delta 0)
To github.com:IITDhCSE/dem0.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```



what does the –u option do?

Git – Releasing Code

- Tagging
  1. Check for unsaved changes in local repository.

```
nikhilh@ndhpc01:~/dem0$ git status .
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

  1. Create a tag and associate a comment with that tag

```
nikhilh@ndhpc01:~/dem0$ git tag -a VERSION1 -m "Release version 1 implements feature XYZ"
```

  2. Save tags in remote repository

```
nikhilh@ndhpc01:~/dem0$ git push --tags
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Writing objects: 100% (1/1), 191 bytes | 95.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To github.com:IITDhCSE/dem0.git
 * [new tag]         VERSION1 -> VERSION1
```

Git – Recap..

```
1. git clone (creating a local working copy)
2. git add (staging the modified local copy)
3. git commit (saving local working copy)
4. git push (saving to remote repository)
5. git tag (Naming the release with a label)
6. git push --tags (saving the label to remote)
```

- Note that commands 2, 3, and 4 are common to Method 1 and Method 2.
- Please read https://git-scm.com/book/en/v2 for details

For git download on Windows: https://git-scm.com/download/win

Makefile or makefile

- Is a file, contains instructions for the `make` program to generate a target (executable).
- Generating a target involves:
  1. Preprocessing (e.g. strips comments, conditional compilation etc.)
  2. Compiling (.c -> .s files, .s -> .o files)
  3. Linking (e.g. making printf available)
- A `Makefile` typically contains directives on how to do steps 1, 2, and 3.

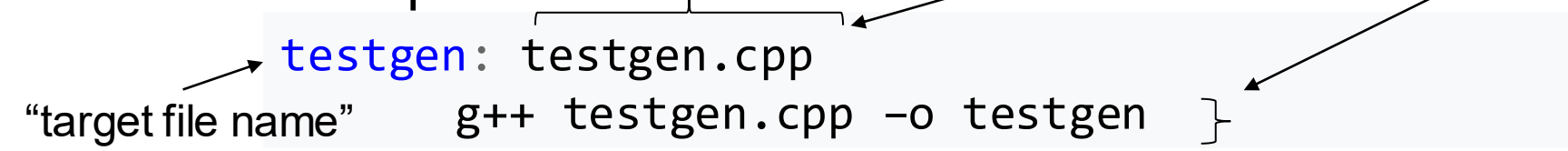
Makefile - Format

1. Contains series of 'rules'-

```
target: dependencies
[TAB] system command(s)
```

Note that it is important that there be a TAB character before the system command (not spaces).

Example:



2. And Macro/Variable definitions -

```
CFLAGS = -std=c++11 -g -Wall -Wshadow --pedantic -Wvla –
Werror

GCC = g++
```

Makefile - Usage

– The ‘make’ command (Assumes that a file by name ‘makefile’ or ‘Makefile’. exists)

```
n2021/slides/week4_codesamples$ cat makefile
vectorprod: vectorprod.cpp scprod.cpp scprod.h
        g++ vectorprod.cpp scprod.cpp -o vectorprod
```

- Run the ‘make’ command

```
n2021/slides/week4_codesamples$ make
g++ vectorprod.cpp scprod.cpp -o vectorprod
```

`Makefile` - Benefits

- Systematic dependency tracking and building for projects
  - Minimal rebuilding of project
  - Rule adding is ‘declarative’ in nature (i.e. more intuitive to read. *caveat: make also lets you write equivalent rules that are very concise and non-intuitive.*)
- To know more, please read: https://www.gnu.org/software/make/manual/html_node/index.html#Top

make - Demo

- Minimal build
  - What if only `scprod.cpp` changes?
- Special targets (`.phony`)
  - E.g. explicit request to `clean` executes the associated recipe. What if there is a file named `clean`?
- Organizing into folders
  - Use of variables (built-in (`CXX, CFLAGS`) and automatic (`$@, $^, $<`))

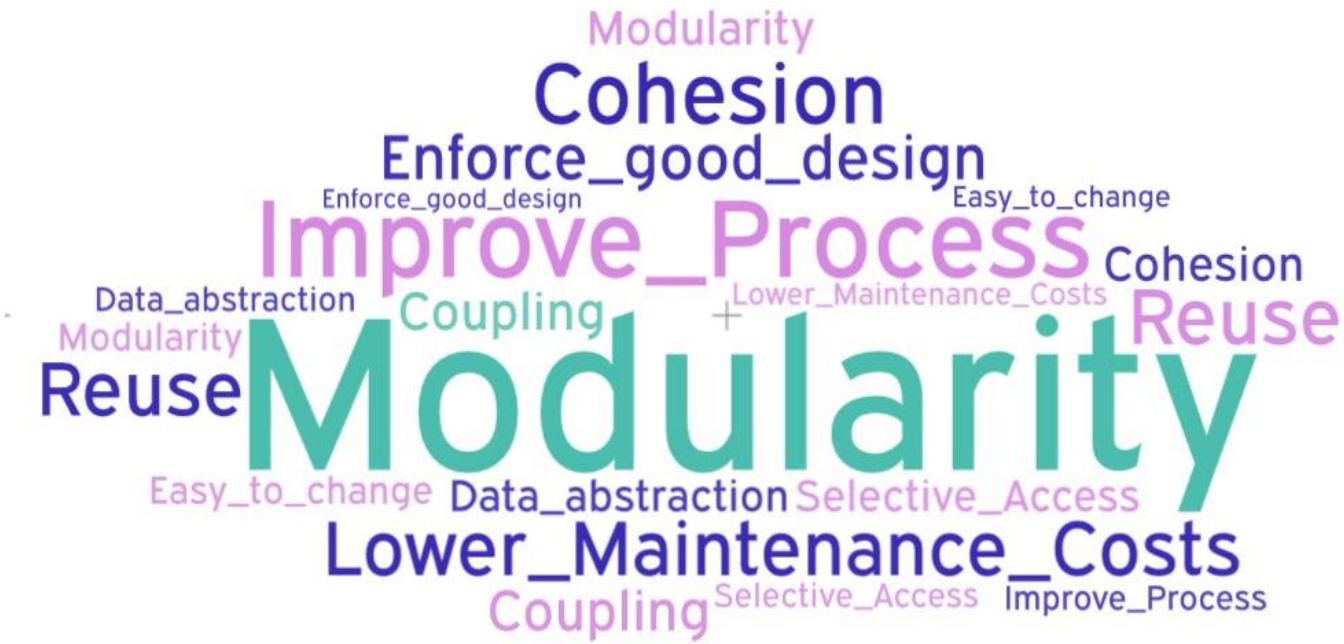
refer to week4_codesamples

Object Orientation

- What does it mean to think in terms of object orientation?
  1. Give precedence to data over functions *(think: objects, attributes, methods)*
  2. Hide information under well-defined and stable interfaces *(think: encapsulation)*
  3. Enable incremental refinement and (re)use *(think: inheritance and polymorphism)*

Object Orientation: Why?

- Improve costs
- Improve development process and
- Enforce good design



Objects and Instances

- Object is a computational unit
  - Has a state and operations that operate on the state.
  - The state consists of a collection of *instance* variables or attributes.
  - Send a "message" to an object to invoke/execute an operation (*message-passing metaphor* in traditional OO thinking)
- An instance is a *specific version* of the object

Classes

- Template or blueprint for creating objects. Defines the shape of objects
  - Has *features* = attributes + operations
  - New objects created are *instances of the class*
  - E.g.

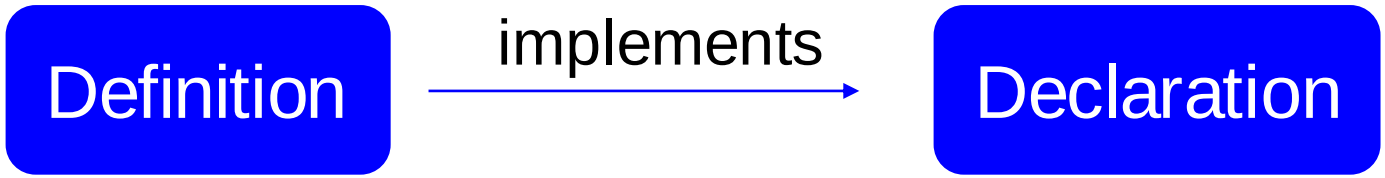
Class - lollypop mould

Objects - lollypops

Classes continued..

- Operations defined in a class are a prescription or service provided by the class to access the state of an object
- Why do we need classes?
  - To define user-defined types / invent new types and extend the language
  - Built-in or Primitive types of a language – int, char, float, string, bool etc. have implicitly defined operations:
    - E.g. cannot execute a *shift* operator on a negative integer
  - Composite types *(read: classes)* have operations that are implicit as well as those that are explicitly defined.

Classes declaration vs. definition



Implementation of functions in a .cpp file

listing of functions and attributes in a .h file

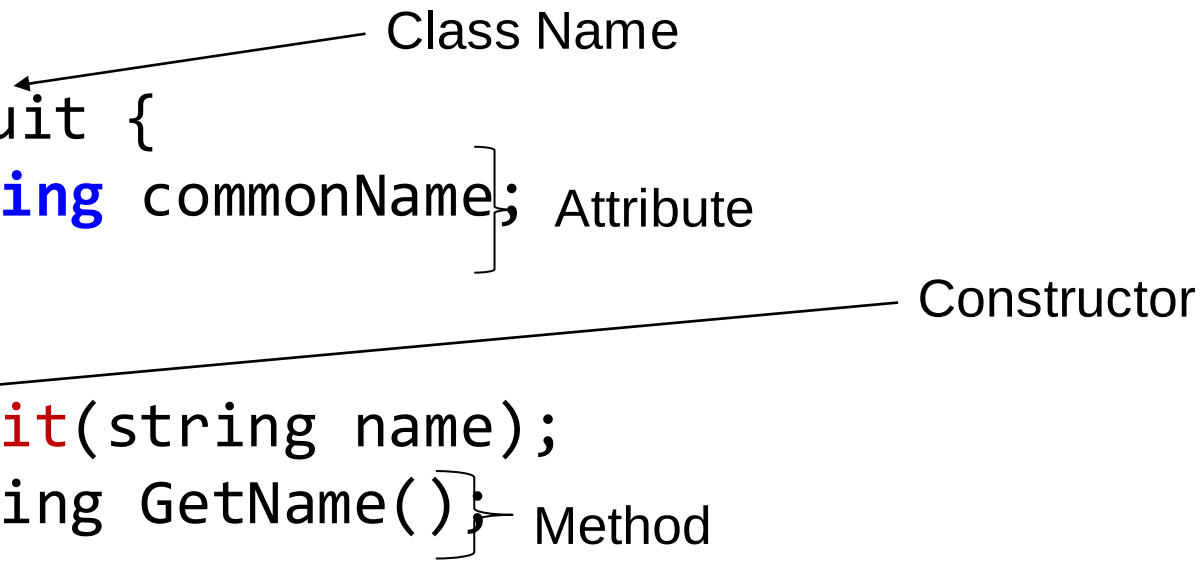
Classes: declaration

- *file* `Fruit.h`

```
#include<string>


class Fruit {
      string commonName;

public:
      Fruit(string name);
      string GetName();
};
```



Trivia: Python doesn't support data hiding

Classes: access control

- Public / Private / Protected

```
class Fruit {
        string commonName; // private by default

public:
        Fruit(string name);
        string GetName();
};
```

- Private: methods-only (self) access
- Public: all access
- Protected: methods (self and *sub-class*) access

Classes: definition

- *file* `Fruit.cpp`

```
#include<Fruit.h>

//constructor definition: initialize all attributes
Fruit::Fruit(string name) {
	commonName = name;
}
//constructor definition can also be written as:
Fruit::Fruit(string name): commonName(name) { }

string Fruit::GetName() {
	return commonName;
}
```

Objects: creation and usage

- *file* `Fruit.cpp`

```
#include<Fruit.h>

Fruit::Fruit(string name): commonName(name) { }
string Fruit::GetName() { return commonName; }

int main() {
        Fruit obj1("Mango"); //calls constructor
        //following line prints "Mango"
        cout<<obj1.GetName()<<endl; //calls GetName
method
}
```

- *How is `obj1` destroyed?* – by calling *destructor*

Objects: Destructor

```
Fruit::~Fruit(){ } //default destructor implicitly
defined

int main() {
        Fruit obj1("Mango"); //statically allocated
object
        Fruit* obj2 = new Fruit("Apple"); //dynamic
object
        delete obj2; //calls obj2->~Fruit();
        //calls obj1.~Fruit()
}
```

- Statically allocated objects: Automatic
- Dynamically allocated objects: Explicit

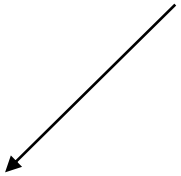
Post-class Exercise - Encapsulation

- The earlier quiz at the beginning of the class was a Pre-class Exercise.
- Re-attempt the same Quiz.

Inheritance

- Create a brand-new class based on existing class

```
file Mango.h
#include<Fruit.h>
class Mango : public Fruit {
        string variety;
public:
        Mango(string name, string var) : Fruit(name),
variety(var){}
};
```



- Fruit is a base type, Mango is a sub-type
- Sub-type inherits attributes and methods of its base type

Inheritance

```
file Fruit.h
#include<string>

class Fruit {
        string commonName;
public:
        Fruit(string name);
        string GetName();
};
```

```
file Mango.h
#include<Fruit.h>
class Mango : public Fruit {
        string variety;
public:
        Mango(string name, string var) :
Fruit(name), variety(var){}
};
```

```
file Fruit.cpp
...
int main() {
        Mango item1("Mango", "Alphonso"); //create sub-class object
        cout<<item1.GetName()<<endl; //only commonName is printed!
}                                    (variety is not included).
```

commonName variety

Refer slide 41.

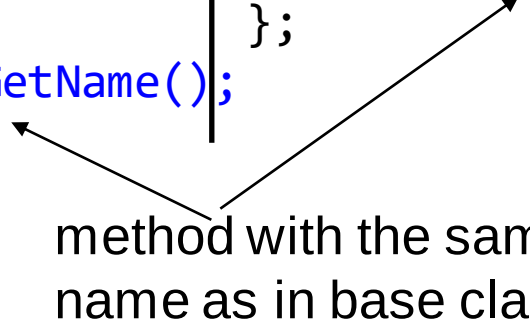
Method overriding

- Customizing methods of derived / sub- class

```
file Fruit.h
#include<string>

class Fruit {
        string
commonName;
public:
        Fruit(string
name);
        string GetName();
};
```

```
file Mango.h
#include<Fruit.h>
class Mango : public Fruit {
        string variety;
public:
        Mango(string name, string var) :
Fruit(name), variety(var){}
    string GetName();
};
```



Method overriding

```
file Fruit.h
#include<string>

class Fruit {
protected:
        string commonName;
public:
        Fruit(string name);
        string GetName();
};
```

```
file Mango.h
#include<Fruit.h>
class Mango : public Fruit {
        string variety;
public:
        Mango(string name, string var) :
Fruit(name), variety(var){}
        string GetName() {    return
commonName + "_" + variety; }
};
```


accessing base class attribute

Method overriding

file Fruit.h

```
#include<string>

class Fruit {
protected:
        string commonName;
public:
        Fruit(string name);
        string GetName();
};
```

file Mango.h

```
#include<Fruit.h>
class Mango : public Fruit {
        string variety;
public:
        Mango(string name, string var) :
Fruit(name), variety(var){}
        string GetName() {    return
commonName + "_" + variety; }
};
```

file Fruit.cpp

```
...
int main() {
        Mango item1("Mango", "Alphonso"); //create sub-class object
        cout<<item1.GetName()<<endl;   //prints "Mango_Alphonso"
}
```

Polymorphism

- Ability of one type to appear and be used as another type
- E.g. type `Mango` used as type `Fruit`

```
file Fruit.cpp
...
int main() {
//create a sub-class object and initialize it to a pointer of
//type base-class
        Fruit* item1 = new Mango("Mango", "Alphonso");
        cout<<item1->GetName()<<endl;  //prints "Mango" !
        ...
}
```

Trivia: Java treats all functions as virtual

Polymorphism

- Declare overridden functions as `virtual` in base class
- Invoke those functions using pointers

```
file Fruit.h
#include<string>

class Fruit {
protected:
        string commonName;
public:
        Fruit(string name);
        virtual string GetName();
};
```

```
file Mango.h
#include<Fruit.h>
class Mango : public Fruit {
        string variety;
public:
        Mango(string name, string
var) : Fruit(name), variety(var){}
string GetName() {    return
commonName + "_" + variety; }
};
```

```
Fruit* item1 = new Mango("Mango", "Alphonso");
cout<<item1->GetName()<<endl; //prints "Mango_Alphonso"
```

Polymorphism and Destructors

- declare base class destructors as `virtual` if using base class in a polymorphic way

```
file Fruit.h
#include<string>

class Fruit {
protected:
        string commonName;
public:
        Fruit(string name);
        virtual string GetName();
        virtual ~Fruit();
};
```

```
...
Fruit* item1 = new Mango("Mango",
"Alphonso");
...
delete item1; //calls Mango::~Mango()
first and then Fruit::~Fruit()
```

Post-class Exercise - Inheritance

- The earlier quiz at the beginning of the class was a Pre-class Exercise.
- Re-attempt the same Quiz.

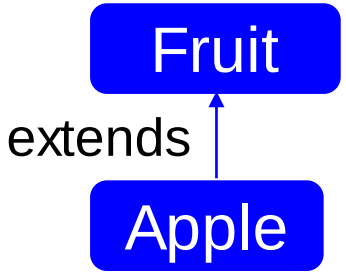
Abstract base classes

- A class can have a `virtual` method without a definition – *pure virtual functions*
- E.g

```
class Fruit {
protected:
        string commonName;
        float weight;
        float energyPerUnitWeight; //in kCals / 100g
public:
        Fruit(string name, float weight);
        virtual string GetName();
        virtual ~Fruit();
        virtual void Energy() = 0;
};
```

Energy is 'pure' – no implementation

Defining pure virtual function

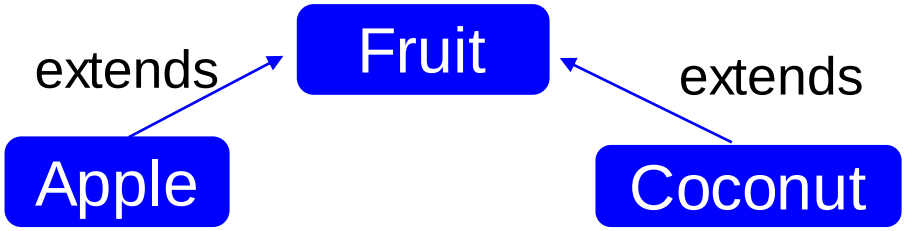



```
class Apple : public Fruit {
        vector<pair<string, float> > constituents;
public:
        Apple(string name, float weight);
        virtual ~Apple();
        . . .
        void Energy() {
        energyPerUnitWeight = ComputeEnergy(weight, constituents);
        }
};
```

Pure virtual method defined in derived class.

Base class attribute

Defining pure virtual function




```
class Coconut : public Fruit {
        vector<pair<string, float> > constituents;
public:
        Coconut(string name, float weight);
        virtual ~Coconut();
        . . .
        void Energy() {
        float effWeight = GetEdibleContentWeight();
        energyPerUnitWeight = ComputeEnergy(effWeight, constituents);
        }
};
```

Computation is different from that of Apple's method

Abstract base classes..

- Cannot create objects from abstract base classes. But may need constructors. Why?

```
Fruit item1; //not allowed. Fruit::Energy() is pure virtual
```

- Can create pointers to abstract base classes and use them in polymorphic way

```
Fruit* item1 = new Apple(“Apple”, 0.24);
cout<<item1->Energy()<<“Kcals per 100 g”<<endl;
```

- Often used to create *interfaces*

Friend functions

- Can access `private` and `protected` members

```
class Coconut {
        vector<pair<string, float> > constituents;
public:
        ...
        friend float ComputeEnergy(float wt, Coconut* c);
};

float ComputeEnergy(float weight, Coconut* c) {
//get a set of items, for each item, get its weight and
//energy_per_g. multiply both. Sum the product of all items...
//read from c->constituents to get the set of items.
}
```

The non-member function `ComputeEnergy` can access private attribute `constituent` of `Coconut` class