

# CS601: Software Development for Scientific Computing

Autumn 2022

Week8: Intermediate C++ (object orientation),  
Motifs: N-body Problems

# Course progress so far:

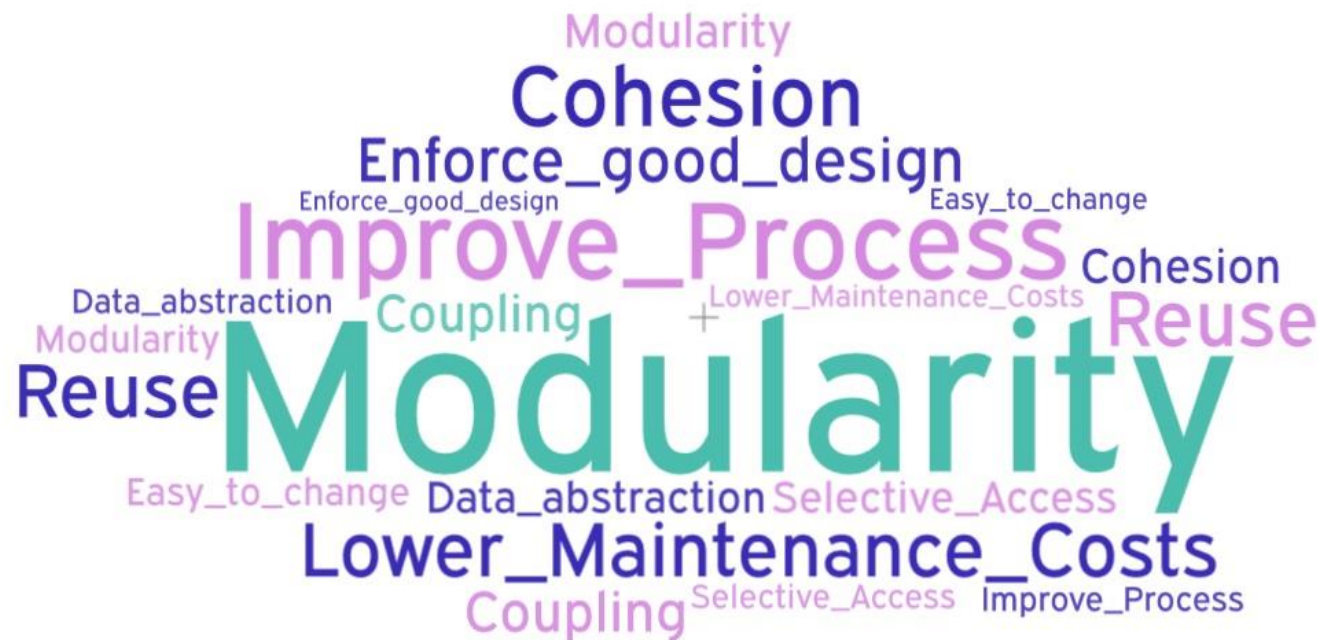
- Computational thinking
  - Data representation (IEEE 754)
  - System Architecture (cache hierarchy, pipelined logic)
  - Language considerations (C/C++ features)
- Patterns / Motifs in Scientific Computing
  - Dense matrix computations, Sparse matrix computations, FFT
- Tools
  - Git, make, overview of compiler tool chain.

# Course in the next 7 weeks:

- Computational thinking
  - Data representation (IEEE 754, [Object-oriented design](#))
  - System Architecture (cache hierarchy, pipelined logic)
  - Language considerations (C/C++ features [Generic programming etc.](#))
- Patterns / Motifs in Scientific Computing
  - Dense matrix computations, Sparse matrix computations, FFT
  - [N-body problems](#), [Structured and Unstructured grids](#)
- Tools
  - Git, make, overview of compiler tool chain.
  - [Doxygen](#), [gdb](#), [valgrind](#), [gprof](#)

# Recap: Object Orientation: Why?

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



© Nikhil Hegde 2020

# Header file (myvec.h)

```
#ifndef MYVEC_H
#define MYVEC_H

-

#endif
```

# Header file (myvec.h)

- Declare the class

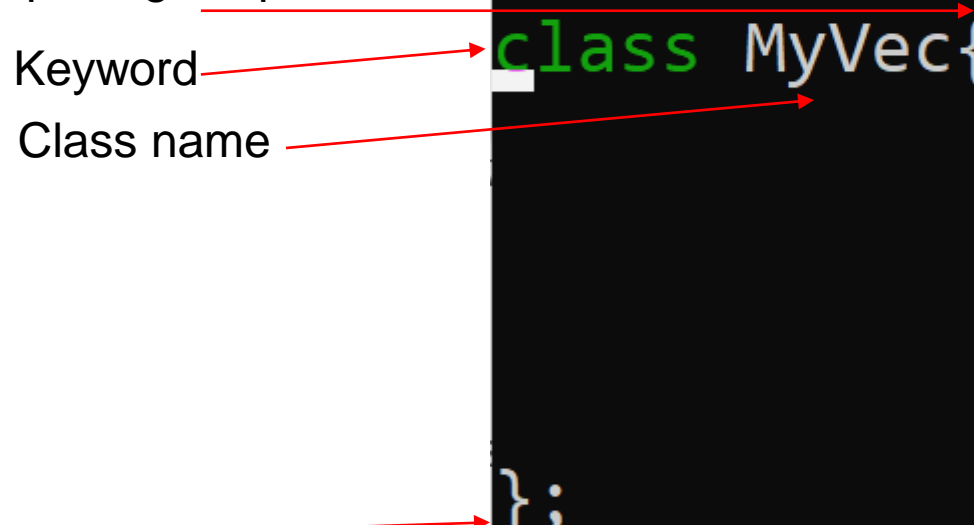
Class *declaration* opening scope

Keyword

Class name

Class *declaration* closing scope

```
#ifndef MYVEC_H
#define MYVEC_H
class MyVec{
:
};
#endif
```

A diagram with red arrows pointing from the text labels to the corresponding parts of the C++ code. An arrow from 'Class declaration opening scope' points to the line containing '#define MYVEC\_H'. An arrow from 'Keyword' points to the word 'class'. An arrow from 'Class name' points to 'MyVec'. An arrow from 'Class declaration closing scope' points to the closing brace '}' of the class definition.

# Header file (myvec.h)

```
#ifndef MYVEC_H
#define MYVEC_H
class MyVec{
    //private attributes
    double* data;
    int vecLen;

};
#endif
```

Declaring attributes



# Header file (myvec.h)

```
#ifndef MYVEC_H
#define MYVEC_H
class MyVec{
    //private attributes
    double* data;
    int vecLen;
public:
    MyVec(int len); //constructor
    ~MyVec(); //destructor
};
#endif
```

Specifying access control



Declaring operations

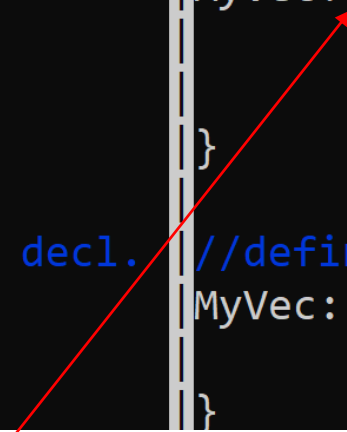




# Defining the class (myvec.h and myvec.cpp)

```
#ifndef MYVEC_H
#define MYVEC_H
class MyVec{
    //private attributes
    double* data;
    int vecLen;
public:
    MyVec(int len); //constructor decl.
    ~MyVec(); //destructor decl.
};
#endif
```

```
#include "myvec.h"
//defining the constructor
MyVec::MyVec(int len) {
    vecLen=len;
    data=new double[vecLen];
}
//defining the destructor
MyVec::~~MyVec() {
    delete [] data;
}
```



Scope resolution operator

Constructor: no return type.

Destructor: no parameters, no return type.

# Defining the class (myvec.h and myvec.cpp)

```
#ifndef MYVEC_H
#define MYVEC_H
class MyVec{
    //private attributes
    double* data;
    int vecLen;
public:
    MyVec(int len); //constructor decl.
    ~MyVec(); //destructor decl.
    int GetVecLen(); //member function
};
#endif
~
~
```

```
#include "myvec.h"
//defining the constructor
MyVec::MyVec(int len) {
    vecLen=len;
    data=new double[vecLen];
}

//defining the destructor
MyVec::~~MyVec() {
    delete [] data;
}

//defining GetVecLen member function
int MyVec::GetVecLen() {
    return vecLen;
}
```

# Using an object

```
#include<iostream>
#include"myvec.h"
using namespace std;
int main() {
    MyVec v(10); //calls the constructor and passes the argument 10
    int size=v.GetVecLen(); //calls the member function
    cout<<"size of MyVec is: "<<size<<" elements"<<endl;
}
```

# Recap: 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()
```

# Exercise

- <https://forms.gle/xzd83oioSmdyTBn86>

# Recap: Abstract base classes

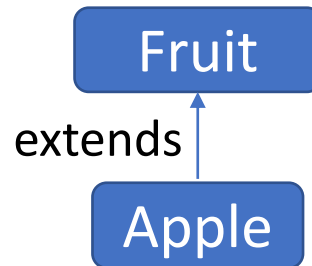
- A class can have a `virtual` method without a definition – *pure virtual functions*

- 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

# Recap: Defining pure virtual function

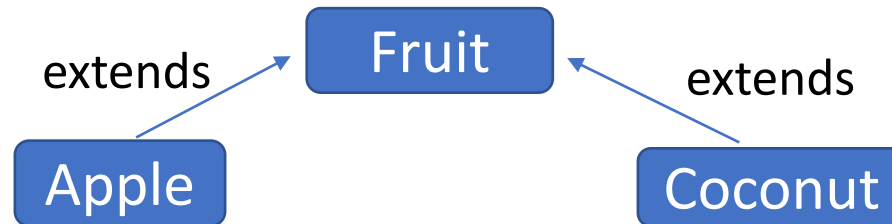


```
class Apple : public Fruit {  
    vector<pair<string, float> > constituents;  
public:  
    Apple(string name, float weight);  
    virtual ~Apple();  
    . . .  
    void Energy() {  
        energyPerUnitWeight = ComputeEnergy(weight, constituents);  
    }  
};
```

Pure virtual method defined in derived class.

Base class attribute

# Recap: 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



# Recap: 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*

# Recap: 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

# Exercise

- <https://forms.gle/JwVF8zSj9Trp4qLx5>

# Operator overloading

- How can we assign one object to another?

```
Apple a1("Apple", 1.2); //constructor Apple::Apple(string, float)  
                        //is invoked
```

```
Apple a2; //constructor Apple::Apple() is invoked.
```

```
a2=a1 //a1 is assigned to a2. assignment operator invoked
```

```
Apple& Apple::operator=(const Apple& rhs){  
    commonName = rhs.commonName;  
    weight = rhs.weight;  
    energyPerUnitWeight = rhs.energyPerUnitWeight;  
    constituents = rhs.constituents;  
    return *this;  
}
```

*Called Copy Assignment Operator*

# Operator overloading []

```
#ifndef MYVEC_H
#define MYVEC_H
class MyVec{
    //private attributes
    double* data;
    int vecLen;
public:
    MyVec(int len); //constructor decl.
    ~MyVec(); //destructor decl.
    int GetVecLen(); //member function
    double& operator[](int index);

};

#endif

    delete [] data;
}

//defining GetVecLen member function
int MyVec::GetVecLen() {
    return vecLen;
}

double& MyVec::operator[](int index) {
    return data[index];
}
```

# Operator overloading - usage

```
#include<iostream>
#include"myvec.h"
using namespace std;
int main() {
    MyVec v(10); //calls the constructor MyVec::MyVec(int) and passes the argument 10
    int size=v.GetVecLen(); //calls the member function
    cout<<"size of MyVec is: "<<size<<" elements"<<endl;
    cout<<"Setting first element to 100"<<endl;
    v[0]=100;
    cout<<"Fetching first element value: "<< v[0] << endl;
}
```

# Copying Objects

```
Apple a1("Apple_red", 0.2);
```

```
Apple a2 = a1; //calls copy constructor
```

```
Apple::Apple(const Apple& rhs) {  
    commonName = rhs.commonName;  
    weight = rhs.weight;  
    energyPerUnitWeight = rhs.energyPerUnitWeight;  
}
```

# Copy constructor – another example

```
#include<iostream>
#include"myvec.h"
using namespace std;
int main() {
    MyVec v(10); //calls the constructor MyVec::MyVec(int) and passes the argument 1
    int size=v.GetVecLen(); //calls the member function
    cout<<"size of MyVec is: "<<size<<" elements"<<endl;
    cout<<"Setting first element to 100"<<endl;
    v[0]=100;
    cout<<"Fetching first element value: "<< v[0] << endl;
    MyVec v2=v; //calls the copy constructor
    cout<<"v2's first element: "<<v2[0]<<endl;
}
```

- Not necessary to define the copy constructor. Compiler defines one for us.



```

#include<iostream>
#include"myvec.h"
using namespace std;
int main() {
    MyVec v(10); //calls the constructor MyVec::MyVec(int) and passes the argument 10
    int size=v.GetVecLen(); //calls the member function
    cout<<"size of MyVec is: "<<size<<" elements"<<endl;
    cout<<"Setting first element to 100"<<endl;
    v[0]=100;
    cout<<"Fetching first element value: "<< v[0] << endl;
    MyVec v2=v; //calls the copy constructor
    cout<<"v2's first element: "<<v2[0]<<endl;
}

```

```

size of MyVec is: 10 elements
Setting first element to 100
Fetching first element value: 100
v2's first element: 100
free(): double free detected in tcache 2
Aborted

```

```

#include<iostream>
#include"myvec.h"
using namespace std;
int main() {
    MyVec v(10); //calls the constructor MyVec::MyVec(int) and passes the argument 10
    int size=v.GetVecLen(); //calls the member function
    cout<<"size of MyVec is: "<<size<<" elements"<<endl;
    cout<<"Setting first element to 100"<<endl;
    v[0]=100;
    cout<<"Fetching first element value: "<< v[0] << endl;
    MyVec v2=v; //calls the copy constructor
    cout<<"v2's first element: "<<v2[0]<<endl;
}

```

```

size of MyVec is: 10 elements
Setting first element to 100

```

*If you don't define a copy constructor, in some cases, e.g., for class MyVec, the program aborts. Why in this case?*

```

v2's first element: 100
free(): double free detected in tcache 2
Aborted

```

# const and references

```
#ifndef MYVEC_H
#define MYVEC_H
class MyVec{
    //private attributes
    double* data;
    int vecLen;
public:
    MyVec(int len); //constructor decl.
    MyVec(const MyVec& rhs); //copy constructor decl.
    int GetVecLen() const; //member function decl.
    double& operator[](int index) const;
    ~MyVec(); //destructor decl.
};

//defining GetVecLen member function
int MyVec::GetVecLen() const {
    return vecLen;
}

double& MyVec::operator[](int index) const {
    return data[index];
}

MyVec::MyVec(const MyVec& rhs) {
    vecLen=rhs.GetVecLen();
    data=new double[vecLen];
    for(int i=0;i<vecLen;i++) {
        data[i] = rhs[i];
    }
}
```

```

#ifndef MYVEC_H
#define MYVEC_H
class MyVec{
    //private attributes
    double* data;
    int vecLen;

public:
    MyVec(int len); //constructor decl.
    MyVec(const MyVec& rhs); //copy constructor decl.

    int GetVecLen() const; //member function decl.
    double& operator[](int index) const; //member function decl.
    ~MyVec(); //destructor decl.
};

}

MyVec::MyVec(const MyVec& rhs) {
    vecLen=rhs.GetVecLen();
    data=new double[vecLen];
    for(int i=0;i<vecLen;i++) {
        data[i] = rhs[i];
    }
}

//defining GetVecLen member function
int MyVec::GetVecLen() const {
    return vecLen;
}

double& MyVec::operator[](int index) const {
    return data[index];
}

```

*Define the copy constructor. Now you need to make changes to other methods (const) as well.*

```

Setting first element to 100
Fetching first element value: 100
v2's first element: 100

```

# Const and References - Summary

- Allow for compiler optimizations
  - pass-by-reference: allows for passing large objects to a function call
- Tell us immediately (by looking at the interface) that a parameter is read-only

# Detour: References and Const

- We saw reference variables earlier (week 2)
  - Closely related to pointers:
  - Directly *name* another object of the *same* type.
  - Recall:
    - A pointer is defined using the \* (dereference operator) symbol.
    - A reference is defined using the & (address of operator) symbol. Furthermore, unlike in pointer definitions, a reference must be defined/initialized with the object that it names (*cannot be changed later*).

# References

```
int n=10;  
int &re=n; //re must be initialized  
int* ptr; //ptr need not be initialized here  
ptr=&n //ptr now initialized (now pointing to n)  
int x=20;  
ptr=&x; //ptr now pointing to x  
re=&x; //is illegal. Cannot change what re names.  
printf("%p %p\n",&re, &n); // re and n are naming the  
same box in memory. Hence, they have the same address.
```

# Quick tour: const

- A type qualifier
- The type is a constant (cannot be modified).
- `const` is the keyword
- Example:

`const int x=10;` //equivalent to: `int const x=10;`

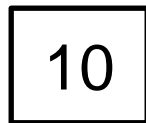
//x is a constant integer. Hence, cannot be modified.

*In what memory segment does x gets stored?*



# Const Properties

- Needs to be initialized at the time of definition
- Can't modify after definition
- `const int x=10;`  
`x=20;` //compiler would throw an error
- `int const x=10;`  
`x=10;` //can't even assign the same value
- `int const y;` //uninitialized const variable y. Useless.



← Can't alter the content of this box

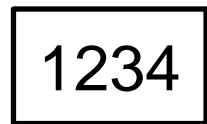
x

# Const Example1 (error)

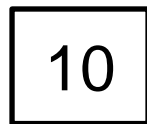
*/\*ptrCX is a pointer to a constant integer. So, can't modify what ptrCX points to.\*/*

```
const int* ptrCX; //or equivalently:  
int const* ptrCX;
```

```
int const x=10;  
ptrCX = &x;  
*ptrCX = 20; //Error
```



ptrCX



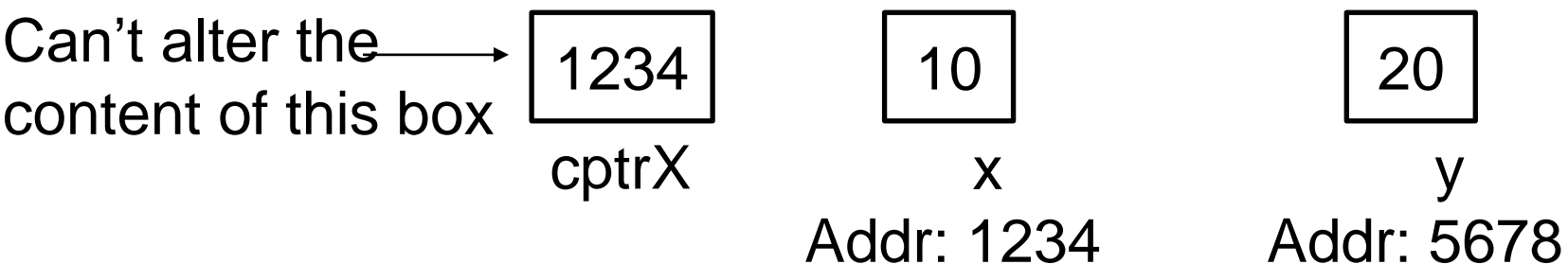
x

Addr: 1234

← Can't alter the content of this box  
using ptrCX or x

# Const Example2 (error)

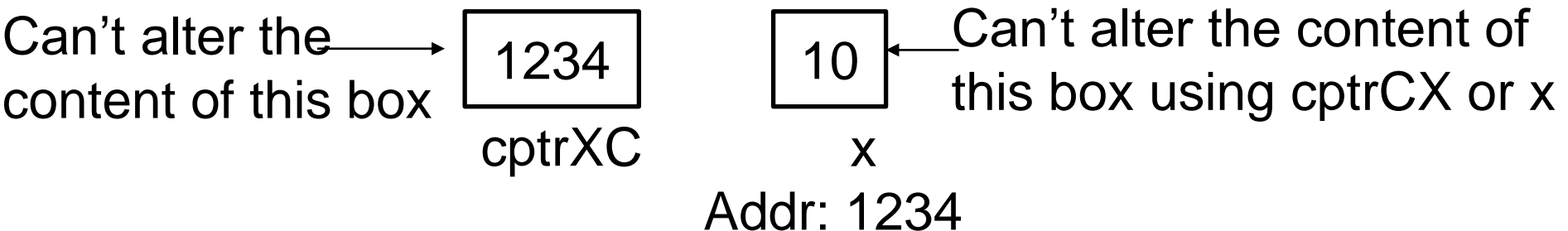
```
/*cptrX is a constant pointer to an integer. So, can't  
point to anything else after initialized.*/  
int x=10, y=20;  
int *const cptrX=&x;  
cptrX = &y; //Error
```



# Const Example3 (error)

*/\*cptrXC is a constant pointer to a constant integer. So, can't point to anything else after initialized. Also, can't modify what cptrXC points to.\*/*

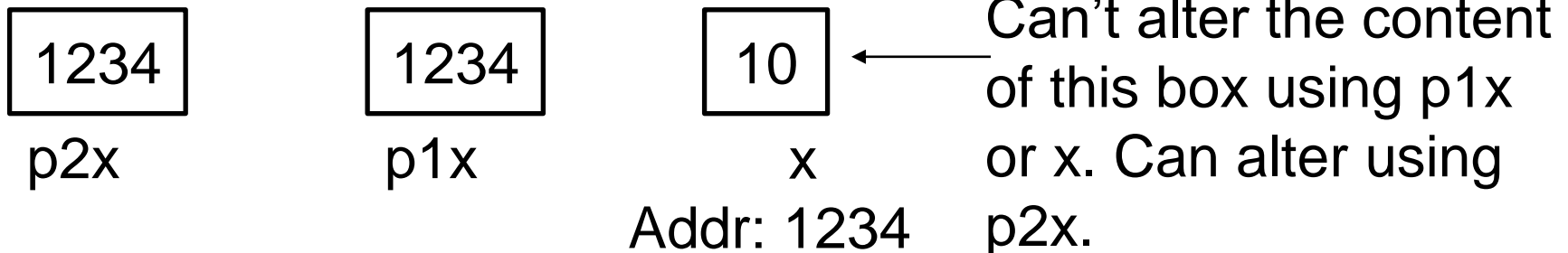
```
const int x=10, y=20;  
const int *const cptrXC=&x;  
int const *const cptrXC2=&x; //equivalent to prev. defn.  
cptrXC = &y; //Error  
*cptrXC = 40; //Error
```



# Const Example4 (warning)

`/*p2x is a pointer to an integer.` So, we can use p2x to alter the contents of the memory location that it points to. However, the memory location contains read-only data - cannot be altered. `*/`

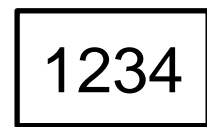
```
const int x=10;
const int *p1x=&x;
int *p2x=&x; //warning
*p2x = 20; //goes through. Might crash depending on memory location accessed
```



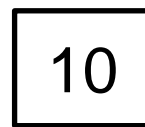
# Const Example5 (no warning, no error)

`/*p1x is a pointer to a constant integer. So, we can't use p1x to alter the content of the memory location that it points to. However, the memory location it points to can be altered (through some other means e.g. using x)*/`

```
int x=10;  
const int *p1x=&x;
```



p1x



x

Addr: 1234

← Can't alter the content of this box using p1x.

Can alter using x.

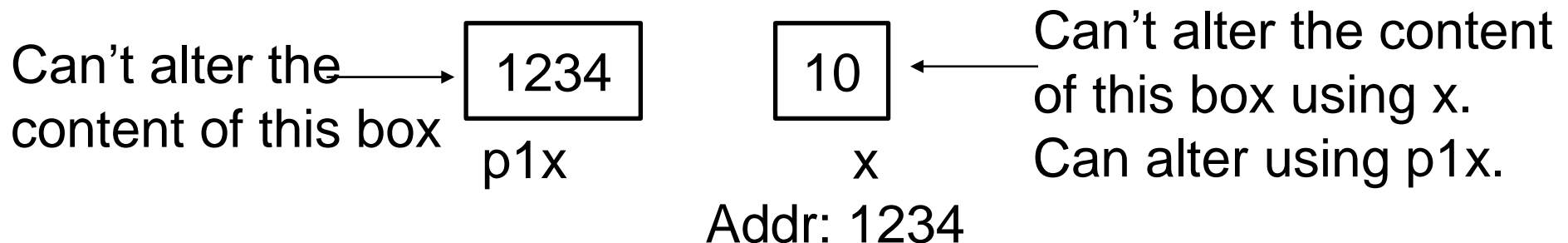
# Const Example6 (warning)

*/\*p1x is a constant pointer to an integer.* So, we can use p1x to alter the contents of the memory location that it points to (and we can't let p1x point to something else other than x). However, the memory location contains read-only data - cannot be altered. \*/

```
const int x=10;
```

```
int *const p1x=&x; //warning
```

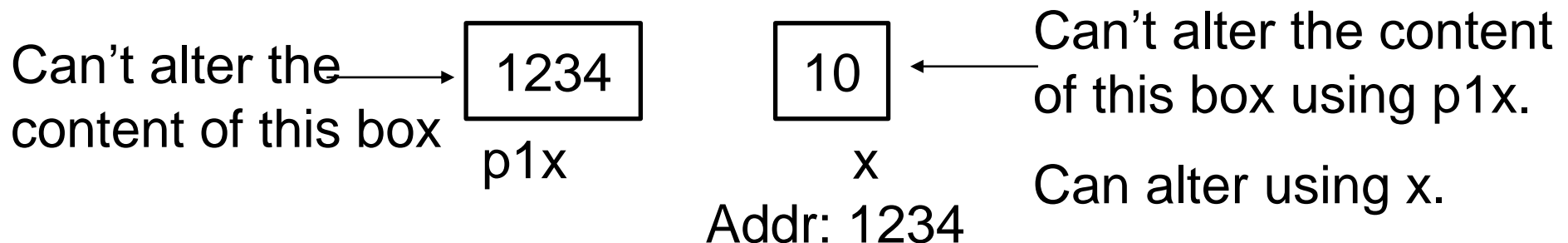
```
*p1x = 20; //goes through. Might crash depending on memory location accessed
```



# Const Example7 (no warning, no error)

*/\*p1x is a constant pointer to a constant integer. So, we can't use p1x to alter the content of the memory location that it points to. However, the memory location it points to can be altered (through some other means e.g. using x)\*/*

```
int x=10;  
const int *const p1x=&x;
```





# Standard Template Library (STL)

- Set of frequently used data structures and algorithms
  - Defined as *parametrized* data types and functions
- E.g.
  - vector, map, queue, pair, sort etc.

# Vectors

- An array that expands and shrinks automatically
  - Parametrized data structure

- E.g.

- `std::vector<int> integers;`

`//empty array that can hold integer numbers`

- `std::vector<Fruit> fruits(10);`

`//array of 10 elements of type Fruit. The 10 objects are initialized by //invoking default constructor`

- Recall that

`class Coconut {`

`vector<pair<string, float> > constituents;`

`...`

Type for a pair of any types (type1, type2)

# Vectors – adding elements

Real-world view  
source:wikipedia

| Coconut meat, raw                    |                     |      |
|--------------------------------------|---------------------|------|
| Nutritional value per 100 g (3.5 oz) |                     |      |
| Energy                               | 354 kcal (1,480 kJ) |      |
| Carbohydrates                        | 15.23 g             |      |
| Sugars                               | 6.23 g              |      |
| Dietary fiber                        | 9.0 g               |      |
| Fat                                  | 33.49 g             |      |
| Saturated                            | 29.698 g            |      |
| Monounsaturated                      | 1.425 g             |      |
| Polyunsaturated                      | 0.366 g             |      |
| Protein                              | 3.33 g              |      |
| Tryptophan                           | 0.039 g             |      |
| Threonine                            | 0.121 g             |      |
| Isoleucine                           | 0.131 g             |      |
| Leucine                              | 0.247 g             |      |
| Lysine                               | 0.147 g             |      |
| Methionine                           | 0.062 g             |      |
| Cystine                              | 0.066 g             |      |
| Phenylalanine                        | 0.169 g             |      |
| Tyrosine                             | 0.103 g             |      |
| Valine                               | 0.202 g             |      |
| Arginine                             | 0.546 g             |      |
| Histidine                            | 0.077 g             |      |
| Alanine                              | 0.170 g             |      |
| Aspartic acid                        | 0.325 g             |      |
| Glutamic acid                        | 0.761 g             |      |
| Glycine                              | 0.158 g             |      |
| Proline                              | 0.138 g             |      |
| Serine                               | 0.172 g             |      |
| Vitamins                             | Quantity            | %DV† |

# Vectors – adding elements

## Object creation and initialization

```
#include<vector> in Fruit.h
```

```
int main() {  
    Coconut* c;  
    c=Coconut("Coconut",1.2)  
    //..  
}
```

```
Coconut::Coconut(string name, float weight) : Fruit(name, weight) {  
    constituents.push_back(make_pair("sugars",6.23));  
    constituents.push_back(make_pair("fiber",9));  
    //...  
}
```

# Vectors – adding elements

## Object layout in memory

### **Fruit part of the object:**

```
commonName = "Coconut"  
Weight = 1.2  
energyPerUnitWeight = 3.6  
vptr = ...
```

### **Coconut part of the object:**

```
constituents = {  
  <sugars, 6.23>,  
  <fiber, 9>,  
  <saturated_fat, 29.69>,  
  <water, 47g>,  
}
```

# Vectors – operations

declaration: `vector<pair<string, float> > constituents;`

Reading elements:

```
constituents.push_back(make_pair("sugars",6.23))  
pair<string, float> tmpVal = constituents[0];
```

Removing elements:

```
constituents.push_back(make_pair("fiber",9))  
constituents.pop_back();
```

Finding number of elements:

```
cout<<constituents.size()<<endl;
```

# Vectors – operations

declaration: `vector<pair<string, float> > constituents;`

Element-wise inspection (iterating over vector elements):

```
vector<pair<string, float>::iterator it;
for(it=constituents.begin(); it!=constituents.end(); it++) {
    pair<string, float> elem = *it;
    cout<<elem.first<<“,”<<elem.second<<endl;
    //can also use cout<<it->first<<“,”<<it->second<<endl;
}
```

Reference: <http://www.cplusplus.com/reference/vector/vector/>

# sort

- Sort fruits by their weight / energy / name

```
#include<algorithm>
bool comp(Fruit* obj1, Fruit* obj2) {
    if(obj1->GetWeight() < obj2->GetWeight())
        return true;
    return false;
}

int main() {
    Apple* a1=new Apple("Apple",0.24);
    Orange* o=new Orange("Orange",0.15);
    Mango* m=new Mango("Mango",0.35);
    Apple* a2=new Apple("Apple",0.2);
    vector<Fruit*> fruits;
    fruits.push_back(a1);
    fruits.push_back(o);
    fruits.push_back(m);
    fruits.push_back(a2);
    sort(fruits.begin(),fruits.end(),comp);
}
```



# Exceptions

- Preferred way to handle logic and runtime errors
  - Unhandled exceptions stop program execution. Handle exceptions and recover from errors.
  - Clean separation between error detection and handling.
- Where to use? often in `public` functions
  - no control over arguments passed
- Are there performance penalties?
  - Mostly not. 'exceptions': memory-constrained devices, real-time performance requirements

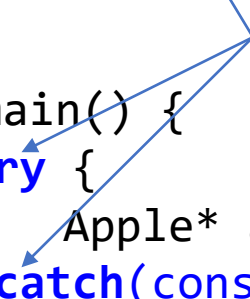
# Exceptions

- E.g.

```
Fruit::Fruit(string name, float wt) {  
    if(wt < 0)  
        throw std::invalid_argument("Invalid weight");  
    ...  
}
```

```
int main() {  
    try {  
        Apple* a = new Apple("Apple_gala",-0.4);  
    } catch(const std::invalid_argument& ia) {  
        cerr<<ia.what()<<endl;  
    }  
}
```

keywords



reference: <http://www.cplusplus.com/doc/tutorial/exceptions/>