

ECE 220

Lecture x0016 - 11/13

Templates & iterators

Recap

- References vs. pointers
- Classes vs. structs
- Inheritance (private/public/protected)
- Constructor in derived classes (**initializer list syntax**).
- Virtual functions
- Pure virtual functions / abstract classes
- Examples of implementations.

Recap: virtual functions

```
#include <iostream>
using namespace std;

class Animal{
public:
    void eat(){
        cout << "I'm eating generic food." << endl;
    }
};
```

```
class Cat : public Animal{
public:
    void eat(){
        cout << "I'm eating a mouse." << endl;
    }
};
```

```
void eat_lunch(Animal *a){
    a->eat();
}
```

```
int main(){
    Animal *anim = new Animal();
    Cat *bruno = new Cat();
    anim->eat();
    bruno->eat();

    eat_lunch(anim);
    eat_lunch(bruno);
}
```

Why didn't Bruno eat a mouse for lunch ?

Need a way for the derived class to **override** the base class function,

... or

We will have to *overload* **eat_lunch** for each new species!

Recap: virtual functions

```
#include <iostream>
using namespace std;

class Animal{
public:
    virtual void eat(){ ←—————
        cout << "I'm eating generic food." << endl;
    }
};

class Cat : public Animal{
public:
    void eat(){
        cout << "I'm eating a mouse." << endl;
    }
};

void eat_lunch(Animal *a){
    a->eat();
}
```

- A virtual function is a member function in the base class that we expect to redefine in derived classes
- What if your colleagues forget to override a virtual function? How to **ensure** it?

Recap: pure virtual functions

Pure virtual functions are used

- if a function doesn't have any use in the base class
- but the function must be implemented by all its derived classes

A pure virtual function doesn't have a function body and it ends with “=0”

```
class Animal{
public:
    virtual void eat()=0;
};

class Cat : public Animal{
public:
    void eat(){
        cout << "I'm eating a mouse." << endl;
    }
};
```

Adding a pure virtual function turns a normal class to an ***abstract*** class!

Recap: abstract class

- **Abstract class** is a class that contains one or more ***pure virtual functions***.
 - No objects of an abstract class can be created/instantiated!
 - A pure virtual function that is not implemented in a derived class **remains** a pure virtual function, so the ***derived class is also an abstract class!***

Recap: Copy constructor

- Last time we implemented a linked list using the `Person` class (node) and `LinkedList` class (linked list).
- Now recall that we could implement a *Stack ADT* with a linked list
 - Push: add at head of linked list
 - Pop: remove from head + *give popped value to caller*
 - How can we do the second part?

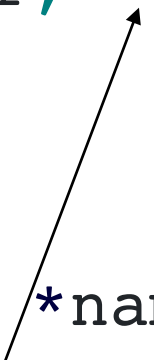
Need a constructor that can generate a new instance of the object from a given instance, i.e. a copy constructor.

Recap: Copy constructor

```
class Person{
    const char *name;
    unsigned int byear;

public:
    Person *next;
    Person(const char *name, unsigned int byear);
    Person(const Person &p);
};
```

Second constructor
useful to copy an
instance of Person.



```
Person::Person(const Person &p){
    this->name = p.name;
    this->byear = p.byear;
    this->next = NULL;
}
```

Called pass by
constant reference.



- **Exercise:** Can we appropriately modify the `LinkedList` class definition and create a derived `Stack` class from it?
- Stack should ***only*** expose the push and pop functions.

Recap: Exercise

- How to modify the `LinkedList` class so that `Stack` can be derived from it?
 - Does `add_at_head` and `del_at_head` need to be public?
 - Can they be private?
 - When popping, we need access to head pointer to call copy constructor - can it still be private?

See: https://github.com/iabraham/ece220-fa25/blob/main/lec21_1111/stack.cpp

Lesson objectives

- Understand and be able to articulate the need for templated functions and classes.
- Be able to implement templated functions and classes.
- Understand the need for friend functions and how to write one.
- Be able to use common templates from the C++ STL (Standard Template Library).

Recall our swap function?

```
void swap(int &a, int &b){  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
void swap(float &a, float &b){  
    float temp = a;  
    a = b;  
    b = temp;  
}
```

```
void swap(char &a, char &b){  
    char temp = a;  
    a = b;  
    b = temp;  
}
```

- Okay, what if you want to swap two floats?
- How about chars?
- Cool, how about two Persons?

```
class Person{  
    const char *name;  
    unsigned int byear;  
    ...  
    ...  
};
```

Are we doomed to keep writing swaps?

Enter C++ templates

- A template is a blueprint for creating a **generic** function or a class.
- A mechanism to allow us to write code once with a *dummy type* (called a template) and then cast to the right kind when needed.

```
int Add(int a, int b){  
    return a+b;  
}
```

```
double Add(double a, double b){  
    return a+b;  
}
```



```
template <typename T>  
T Add(T a, T b){  
    return a+b;  
}
```

Example

```
#include <iostream>
using namespace std;

template <typename T>
T Add(T a, T b){
    return a+b;
}

int main(){
    cout<<Add(1, 3)<<endl;
    cout<<Add(1.2, 3.5)<<endl;
}
```

Well ... what if we want to be able to add 2 to 'C' and get "E"?

You can specify more than one *typename*.

```
template <typename T1, typename T2>
T2 Add(T1 a, T2 b){
    return a+b;
}
```

Exercise

Implement `myswap` so it works for any type of argument. Then use it to swap two instances of `Person`.

```
class Person{
    const char *name;
    unsigned int byear;

public:
    Person *next;
    Person(const char *name, unsigned int byear);
    Person(const Person &p);
};

Person::Person(const Person &p){
    this->name = p.name;
    this->byear = p.byear;
    this->next = NULL;
}
```

Note: It cannot be named `swap`, that will conflict with a templated `swap` function in the standard library.

Class templates

- Just like we can have function templates, we can also have class template.
- Here is a generic node.
- Implement a linked list on this and test with chars and ints

```
#include <iostream>
using namespace std;

template <typename T>
class Node{
    T data;
public:
    Node<T> * next;
    Node(T inval){
        data = inval;
        next = NULL;
    }
    void print(){ cout<<data; }
};
```



```
template <typename H>
void LinkedList<H>::print_list(){
...
}
```

Class templates

- Just like we can have function templates, we can also have class template.
- Here is a generic node.
- Implement a linked list on this and test with chars and ints

```
template <class H>
class LinkedList{
    H *head;

public:
    LinkedList(){
        this->head = NULL;
    }
    void print_list();
    void add_at_head(H &p);
    void del_at_head();
    ~LinkedList();
};
```

What would you need to make this work with our Person class?

Exercise: Make [person_exercise.cpp](#) work.

Advanced operator overloading

- Suppose we want to be able to do:

`std::cout<<object`

where object is an instance of our own class, like `Person`.
Can we?

```
#include <iostream>
using namespace std;

template <typename T>
class Node{
    T data;
public:
    Node<T> * next;
    Node(T inval){
        data = inval;
        next = NULL;
    }
    void print(){ cout<<data; }
};
```

Overloading the << operator

- We need to augment the class definition with a **friend** function.

```
class Person{
    const char *name;
    unsigned int byear;

public:
    Person *next;
    Person(const char *name, unsigned int byear);
    Person(const Person &p);

};

ostream& operator<<(ostream& os, const Person& p){
    os << "(" <<p.name << ", " <<p.byear<< ") ";
    return os;
}
```

Overloading the << operator

- We need to augment the class definition with a friend function.

```
class Person{
    const char *name;
    unsigned int byear;

public:
    Person *next;
    Person(const char *name, unsigned int byear);
    Person(const Person &p);
    friend ostream& operator<<(ostream& os, const Person& p);
    void print(){
        cout<<(*this);
    }
};

ostream& operator<<(ostream& os, const Person& p){
    os << "(" <<p.name <<", " <<p.byear<<") ";
    return os;
}
```


OOP - rule of three

```
class demo{  
    ...  
public:  
    demo(int x); // constructor  
    demo();      // default constructor  
    demo(const demo &x); // copy constructor  
    ~demo();     // destructor  
    demo &operator=(const demo &x); //copy assignment operator  
}
```

If you write one of these for your classes, you probably need to write the other two as well!

https://en.cppreference.com/w/cpp/language/rule_of_three

C++ STL: Standard Template Library

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as *lists, stacks, arrays, etc.*
- STL has five components
 - Algorithms
 - Containers
 - Iterators
 - Functors
 - Adaptors

Left for later classes ←

Algorithms

- STL contains standard and vetted implementations of algorithms for sorting, searching, partitioning, etc.

```
#include <algorithm>
#include <iostream>

using namespace std;

void show(int a[], int array_size){
    int i=0;
    for (i = 0; i < array_size-1; ++i)
        cout << a[i] << ", ";
    cout<<a[i]<<endl;
}
```

```
int main(){
    int a[] = { 1, 5, 8, 9, 6, 7, 3, 4, 2, 0 };
    int asize = sizeof(a) / sizeof(a[0]);
    cout << "The array before sorting is: \n";
    show(a, asize);

    sort(a, a + asize);
    cout << "\n\nThe array after sorting is:\n";
    show(a, asize);
    return 0;
}
```

Containers

- Vectors
 - Dynamically sized but also contiguously stored
 - Fast traversal
 - Insertion at beginning expensive, end ... *variable*
- Lists
 - Doubly linked lists
 - Non-contiguously stored
 - Slower traversal
 - Insertion/deletion constant time

There are many more, but we will talk about these two and deal with rest on need-to-know basis.

Vectors - common operations

- `push_back` – It push the elements into a vector from the back
- `pop_back` – It is used to pop or remove elements from a vector from the back.
- `insert` – It inserts new elements before the element at the specified position
- `assign` – It assigns new value to the vector elements by replacing old ones
- `swap` – It is used to swap the contents of one vector with another vector of same type. Sizes may differ.
- `clear` – It is used to remove all the elements of the vector container
- `front` – Returns a reference to the first element in the vector
- `back` – Returns a reference to the last element in the vector
- `size` – Returns size of the container

Example

```
#include <iostream>
#include <vector>

using namespace std;

int main(){
    vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    cout << "Size: " << g1.size() << endl;
    cout << "Elements: ";

    for (int i = 0; i < 5; i++)
        cout<<g1[i]<<" ";

    cout<<endl;
    return 0;
}
```

This is traditionally how we have been taught to iterate over an array.

But there are many containers in STL: vector, list, queue, map, set, etc.

Need a consistent way to iterate over containers regardless of functionality!

Iterators - common methods

- `begin()` – Used to return the beginning position of the container.
- `end()` – Used to return the position after the end of the container.
- `advance(itr, num)` – Used to increment the iterator `itr` position till the specified number `num`.
- `next(itr, num), prev(itr, num)` - Used to return **new iterators** after incrementing or decrementing `itr` by `num` positions.

Iterators

- Iterators point to the address of elements of a container.

```
#include<iostream>
#include<iterator> // for iterators
#include<vector>    // for vectors

using namespace std;

int main() {
    vector<int> ar = { 1, 2, 3, 4, 5 };
    vector<int>::iterator ptr; // Declaring iterator to a vector

    cout << "The vector elements are : ";
    for (ptr = ar.begin(); ptr < ar.end(); ptr++)
        cout << *ptr << " ";

    return 0;
}
```

Vectors - More operations

- `begin()` – Returns an **iterator** pointing to the first element in the vector
- `end()` – Returns an **iterator** pointing to the theoretical element after last
- `rbegin()` – Returns a reverse **iterator** pointing to the last element in the vector
- `rend()` – Returns a reverse **iterator** pointing to the theoretical element before the first
- `cbegin()` – Returns a **constant** iterator pointing to the first element in the vector.
- `cend()` – Returns a **constant** iterator pointing to the element after last
- `crbegin()` – Returns a **constant** reverse iterator pointing to the last element in the vector
- `crend()` – Returns a **constant** reverse iterator pointing to the theoretical element before the first

Lists - common operations

- `front()` – Returns the value of the first element in the list.
- `back()` – Returns the value of the last element in the list.
- `push_front()` – Adds a new element at the beginning of the list.
- `push_back()` – Adds a new element at the end of the list.
- `pop_front()` – Removes the first element of the list
- `pop_back()` – Removes the last element of the list
- `insert()` – Inserts new elements in the list before the element at a specified position.
- `size()` – Returns the number of elements in the list.

Example

```
#include <iostream>
#include <iterator>
#include <list>
using namespace std;

template <typename T>
void showlist(list<T> g){
    typename list<T>::iterator it;
    for (auto it = g.begin(); it != g.end(); ++it)
        cout << '\t' << *it;
    cout << endl;
}
```

New keyword introduced in C++11,
allows compiler to *deduce* the type.

```
int main(){

    list<int> gqlist1, gqlist2;

    for (int i = 0; i < 10; ++i) {
        gqlist1.push_back(i * 2);
        gqlist2.push_front(i * 3);
    }

    cout << "\nList 1 (gqlist1) is : ";
    showlist(gqlist1);

    cout << "\nList 2 (gqlist2) is : ";
    showlist(gqlist2);

    cout << "\ngqlist2.sort(): ";
    gqlist2.sort();
    showlist(gqlist2);

    return 0;
}
```