ECE 220

Lecture x0015 - 11/12

C++ - Inheritance, polymorphism

Recap

- **We talked about**
  - C vs. C++ obvious differences
  - Default arguments
  - Dynamic allocation
  - Function & operator overloading
  - Structs vs. classes
  - TODO: LinkedList example
- **Announcements**
  - Quiz next week
  - Final exam details now on course website.

Default arguments

```
float bmi_si(float hcm, float kg){
    return kg / (hcm/100 * hcm/100);
}

float bmi_usa(float hin, float lbs){
    return lbs / (hin * hin) * 703;
}
```

C: Write two functions and use appropriate one depending on units at hand.

C++: Write one function which can accept an optional flag for the rare case an European reports their weight and height in centimeters and kilograms

```
float bmi(float ht, float wt, bool si=false){
    float val = wt/(ht*ht);
    if (si)
        return val*10000;
    else
        return val*703;
}
```

Default value is false

Dynamic allocation in C & C++

C	C++
Dynamic allocation is accomplished by `malloc`	Dynamic allocation is accomplished by `new`
Deallocation accomplished by `free`	Deallocation accomplished by `delete`
Both `malloc` and `free` are library functions	Both `new` and `delete` are keyword/operators

```
# include <iostream>

int main(){
  int *p;

  // Allocating an integer's worth of space
  p = new int;

  .
  .
  .
  // Deallocating
  delete p;
}
```

Function overloading

- C++ allows multiple functions with the same name but **different** parameters.

- <u>**Note**</u>: The return value cannot be different

```
double volume(float r){
  return 22.0/7*r*r*r*4/3;
}

double volume(float r, float l){
  return 22.0/7*r*r*l;
}

double volume(float w, float h, float l){
  return width * height * length;
}
```

```
struct student{
    char name[74];
    unsigned long UIN;
    unsigned int year;
    float GPA;
};
```

Structs vs. classes

```
class Student{
    char name[74];
    unsigned long UIN;
    unsigned int year;
    float GPA;

public:
  Student(char const *name, unsigned int UIN,
          unsigned int year, float GPA);

  float get_GPA();
  char const * get_name();
  void set_GPA(float gpa);
};
```

```
Student::Student(char const *name,
                 unsigned int UIN,
                 unsigned int year,
                 float GPA){
  strcpy(this->name, name);
  this->UIN = UIN;
  this->year = year;
  this->GPA = GPA;
}

float Student::get_GPA(){
  return this->GPA;
}

char const * Student::get_name(){
  return this->name;
}

void Student::set_GPA(float gpa){
  this->GPA = gpa;
}
```

Operator overloading

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

class Complex{
  double real;
  double imag;

public:
  Complex(double real, double imag){
    this->real = real;
    this->imag = imag;
  }

  void print(){
    cout<<"(" <<this->real<<" + "<<this->imag<<")";
  }

 Complex operator+(Complex c){
    return Complex(this->real + c.real, this->imag + c.imag);
  }
};
```

```
int main(){
  Complex c1 = Complex(2, 4);
  Complex c2 = Complex(3, -5);
  Complex c3 = c1 + c2;
}
```

New feature: references

```
int val = 10;     // normal variable
int *ptr = &val; // & to get address, * to indicate pointer
int &ref = val;  // & to declare reference to val
```

- Reference is yet another addition to the C/C++ zoo.
- **<u>Key difference</u>**: A pointer is *still a variable* that takes up memory whereas a reference need not (C++ standard leaves it unspecified).
    - Think of it as an *alias* for a variable.
- If you remember the key difference then rest of the behavior is logical.

Pointers vs. references

	Pointer	Reference
Memory address	Has memory allocated for it	May not have memory allocated for it
Function	Stores the memory address of variable	Acts as an *alias* for a variable
Initialization/ reassignment	Can be declared, initialized and also reassigned	Initialized on declaration and cannot be reassigned
Null value	Can be assigned the NULL pointer	Cannot be assigned a NULL value
Dereferencing	Must use the * operator	Automatically dereferenced
Arrays	Can have array of pointers	Cannot create array of references

https://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp4_PointerReference.html

Examples

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

int main(){
  int val = 10;
  int *ptr = &val; // & to get address
  int &ref = val;  // & to declare reference

  cout<<"val  = "<<val<<endl;
  cout<<"*ptr = "<<*ptr<<endl;
  cout<<"ref  = "<<ref<<endl;

  ref = 20;
  cout<<endl<<"val = "<<val<<endl;

  val = 30;
  cout<<"ref = "<<ref<<endl;

  cout<<"ptr = "<<ptr<<endl;
  ptr = &ref;
  cout<<"ptr = "<<ptr<<endl;
}
```

What will be the output?

Which variable(s) changed here?

What about here?

Are these addresses same or different?

Why references when have pointers?

- Mostly safety:
  - No such thing as reference arithmetic & cannot reassign references (can do both to pointers).
  - *Paradigm:* Use references for most use cases and use pointers only when you *must*.
- Passing around large objects to/via functions is simplified (for the programmer) with references:
  - Example *later:* copy constructors

💀 **Advanced Usage** 💀 : `static_cast<void (*)(int &, int&)>(&swap)(val1, val2);`

Examples

Can fail for uninitialized, dangling, or ill-formed pointers!

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

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

Less can go wrong with this version.

```
int main(){
  int val1, val2;
  val1 = 10, val2 = 20;

  cout<<"val1 = "<<val1<<endl;
  cout<<"val2 = "<<val2<<endl;

  swap(&val1, &val2);
  cout<<endl<<"val1 = "<<val1<<endl;
  cout<<"val2 = "<<val2<<endl;

  swap(val1, val2);
  cout<<endl<<"val1 = "<<val1<<endl;
  cout<<"val2 = "<<val2<<endl;
}
```

Which function is called?

Which function is called?

***Overload resolution* fails!**

Solution: Explicit casts

What happens now?

Exercise using classes

- Implement our old linked list using:

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

public:

  Person *next;
  Person(const char *name, unsigned int byear){
    this->name = name;
    this->byear = byear;
    this->next = NULL;
  }
};
```

These are private, if we want to be able to print our linked list will need to implement a print function.

Exercise using classes

- Implement our old linked list using:

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

public:

  Person *next;
  Person(const char *name, unsigned int byear){
    this->name = name;
    this->byear = byear;
    this->next = NULL;
  }

  void print(){
      cout<< "(" << this->name << ", " << this->byear << ")" <<endl;
  }
};
```

Exercise using classes

- How to maintain head pointer, and add/remove functions?
  - Adopt the OOP way

```
class LinkedList{
Person *head;

public:
  LinkedList(){
    this->head = NULL;
  }



};
```

- Basic functions to implement for a linked list?
  - Function to print list
  - Function to add at head
  - Function to remove from head

See Gitlab for full implementation!

What about a class `Hamster` which squeaks?

Inheritance

```
class Dog{
  const char *name;
  int breed;
  int age;
  bool nail_clip;

public:
  Dog(const char *n, int b, int a){
    name = n, breed = b; age = a;
  }

  void greet(const char *p){
    cout<<name<<": Hi, "<<p<<endl;
  }

  void sleep(){
    cout<<name<<": Zzzzzz"<<endl;
  }

  void speak(){
    cout<<name<<": Woof!"<<endl;
  }
};
```

```
class Cat{
  const char *name;
  int breed;
  int age;


public:
  Cat(const char *n, int b, int a){
    name = n, breed = b, age = a;
  }

  void greet(const char *p){
    cout<<name<<": Hi, "<<p<<endl;
  }

  void sleep(){
    cout<<name<<": Zzzzzz"<<endl;
  }

  void speak(){
    cout<<name<<": Meow!"<<endl;
  }
};
```

Inheritance

C++ allows us to define a class based on an existing class, and the new class will inherit members of the existing class.

- The **existing** class - **Base class**
- The **new** class - **Derived class**

Exceptions in inheritance (things not inherited):

- Constructors, destructors of the base class
- Overloaded operators of the base class
- Friend functions of the base class

Inheritance

Base class

```
class Animal{
  const char *name;
  int breed;
  int age;

public:
  Animal(const char *n, int b, int a){
    name = n;
    breed = b;
    age = a;
  }

  void greet(const char *p){
    cout<<name<<": Hi, "<<p<<endl;
  }

  void sleep(){
    cout<<name<<": Zzzzzz"<<endl;
  }

  const char* get_name(){
    return name;
  }
};
```

Inheritance mode

Derived class **Base class**

Derived class

```
class Dog: public Animal{
  bool nail_clip;

public:
  void speak(){
    cout<<get_name()<<": Woof!"<<endl;
  }
};
```

Derived class

```
class Cat: public Animal{
public:

  void speak(){
    cout<<get_name()<<": Meow!"<<endl;
  }
};
```

Inheritance rules

Inheritance	Derived class has access to ….		
	private members	**public members**	**protected members**
Private inheritance	No	No (inherited as private variables)	Yes (inherited as private variables)
Public inheritance	No	Yes (inherited as public variables)	Yes
Protected inheritance	No	Yes (inherited as protected variables)	Yes

Derived class constructor?

```
class Dog: public Animal{
  bool nail_clip;

public:
  Dog(const char *n, int b, int a, bool c){
    nail_clip = c;
  }
  void speak(){
    cout<<get_name()<<": Woof!"<<endl;
  }
};

class Cat: public Animal{
public:

  Cat(const char *n, int b, int a){
  };

  void speak(){
    cout<<get_name()<<": Meow!"<<endl;
  }
};
```

How will `Dog` and `Cat` set their breed, name and age which are part of the `Animal` class and its private members?

Derived class constructor?

```
class Dog: public Animal{
  bool nail_clip;

public:
  Dog(const char *n, int b, int a, bool c) : Animal(n, b, a){
    nail_clip = c;
  }
  void speak(){
    cout<<get_name()<<": Woof!"<<endl;
  }
};

class Cat: public Animal{
public:

  Cat(const char*n, int b, int a) : Animal(n, b, a){
  };

  void speak(){
    cout<<get_name()<<": Meow!"<<endl;
  }
};
```

Will make sure to call the base class constructor first.

It is called *member initializer list* syntax!

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!

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?

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!

Abstract class

- **Abstract class** is a class that contains one or more ***pure virtual functions.***
  - No objects of that abstract class can be created
  - 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
  - An abstract class is intended as an interface to objects accessed through pointers and references (e.g. `eat_lunch` function)

Copy constructor

- 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?

Copy constructor

```
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;
}
```

Second constructor useful to copy an instance of Person.

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.

Exercise - time permitting

- How to modify the `LinkedList` class?
  - 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?

Friend functions

- What if we would like a ***select*** few functions to have access to the objects members?
  - C++ lets you define ***friend*** functions in a class declaration.
  - These classes have access to all class members but are **not** class members themselves

```
class Box {
   double width;

   public:
      friend void printWidth( Box box );
      void setWidth( double wid);
};

// Member function definition
void Box::setWidth( double wid) {
   width = wid;
}

/* Note: printWidth() is not a member
function of any class */
void printWidth( Box box ) {
   /* Because printWidth() is a friend of Box,
      it can directly access any member of this
      class */
   cout << "Width of box : " << box.width <<endl;
}
```