

CS2311 Computer Programming

LT12 Object Oriented Programming-II

Computer Science, City University of Hong Kong

Semester B 2022-23

Review

- Prerequisite: C-like struct and overload
- Class and objects: basic concepts and syntax
- Constructors and destructors

Review: Definition of *struct*

- A *composite data type* that groups a list of variables (possibly different types) under one name
- Variables are stored in a continuous memory areas
- Syntax and example:

```
struct typename {  
    type1  member_var1;  
    type2  member_var2;  
    ...  
};
```

```
struct StudentRecord {  
    char    name[51];  
    char    sid[9];  
    float   GPA;  
};
```

Review: struct initialization

- No memory is allocated when you *define* a struct
- When you declare a variable of a given struct type, enough memory is allocated for storing all struct members *contiguously*
- Example:

```
StudentRecord dannyVariable = {"Danny", "50123456", 80};
```

Review: Accessing Individual Members

- A member variable can be accessed with the use of the dot operator “.”

```
peter.final += 10;
```

- Structure types can have the same member name without conflict

```
struct CS2311Student {  
    char    sid[9];  
    float   asg[3];  
    float   lab[10];  
    float   midterm;  
    float   final;  
};
```

```
struct CS6789Student {  
    char    sid[9];  
    float   asg[5];  
    float   final;  
};
```

```
CS2311Student peter;  
cin >> peter.final;  
CS6789Student danny;  
cin >> danny.final;
```

Review: Example

```
struct CS2311Student {  
    int    sid;  
    float  quiz;  
    float  asg1;  
    float  asg2;  
};
```

```
int main() {  
    CS2311Student sr;  
    cout << "Please enter your id, quiz, a1, and a2 marks\n";  
    cin >> sr.id;  
    cin >> sr.quiz;  
    cin >> sr.asg1;  
    cin >> sr.asg2;  
    cout << sr.id << " cw:" << (sr.quiz+sr.asg1+sr.asg2)/3 << endl;  
    return 0;  
}
```

Review: Struct Pointer

- Struct pointer stores the memory address of the first byte of a struct variable

```
Date d;  
d.year = 2022;  
d.month = 11;  
d.day = 7;  
Date *dPtr = &d;
```

Address	Value	
0xa12	2022	d.year
0xa16	11	d.month
0xa1a	7	d.day
0xa1e	0xa12	dPtr

```
cout << dPtr; // will output "0xa12"
```

Review: Structure Pointer: Arrow Syntax

- Arrow syntax `->`: access structure members using pointer
- Example

```
Date d;  
d.year=2022; d.month=11; d.day=7;  
  
Date *dPtr = &d;  
cout << dPtr->year << " " << dPtr->month << " " << dPtr->day << endl;  
  
dPtr->day++;  
dPtr->month-=2;  
cout << dPtr->day << endl;
```


Review: Function Overload

- *Overloading*: two or more functions with the *same name* but *different implementations*
- Two or more functions are said to be overloaded if they differ in
 - the number of arguments, OR
 - the type of arguments, OR
 - the order of arguments
- When an overloaded function is called, the compiler determines the most appropriate call by comparing function argument types

Review: Overload: Common Errors

```
int sum(int x, int y) {  
    return x+y;  
}  
int sum(int a, int b) {  
    return x+y;  
}  
int main() {  
    int a, b;  
    cin >> a >> b;  
    cout << sum(a, b);  
    return 0;  
}
```

```
int sum(int x, int y) {  
    return x+y;  
}  
char sum(int x, int y) {  
    return '0'+(char)(x+y);  
}  
int main() {  
    int a, b;  
    cin >> a >> b;  
    char s = sum(a, b);  
    cout << s;  
    return 0;  
}
```

Review: Overload: Ambiguous Call

- **Ambiguous call:** when the compiler is unable to choose between two correctly overloaded functions
- Automatic type conversions are the main cause of ambiguity

```
void printData(double x) {  
    cout << "Print double: " << x << endl;  
}  
void printData(float x) {  
    cout << "Print float: " << x << endl;  
}  
int main() {  
    char a = '0';  
    printData(a);    printData((double)a);  
    return 0;  
}
```

Review: Class and Objects

- A *class* is a user-defined data type used as a template for creating *objects*
- For example
 - class: Politician objects: Trump, Biden, Obama
 - class: Country objects: USA, China, India, ...
- A class typically contains:
 - *data fields*: member variables that describe object state (i.e., object attributes or properties)
 - *methods*: member functions that operate on the object (e.g., alter or access object state)

Review: Example

```
char    *body_color;  
char    *eye_color;  
float    pos_x, pos_y;  
float    orient;  
float    powerLevel;  
Camera    eye;  
Speaker    mouth;  
Mic        ear;
```

```
void    start();  
void    shutdown();  
void    moveForward(int step);  
void    turnLeft(int degree);  
void    turnRight(int degree);  
void    listen(Audio *audio);  
Audio    speak(char *str);
```

class: **Robot**

Member
variables

Member
functions

```
Robot eve, wall_e;  
eve.body_color = "White";  
eve.eye_color = "Blue";  
wall_e.body_color = "Yellow";  
wall_e.eye_color = "Black";  
...
```



Review: Object-Oriented Programming (OOP)

- Conventional procedural programming:
 - A program is divided into small parts called functions
 - Focus on solving a problem step by step
- Object-oriented programming
 - A program is divided into objects, each contains data and functions that describe properties, attributes, and behaviours of the object
 - Focus on modelling object interactions in real-world
 - Code reuse, modularity and flexibility, efficient for large projects
 - However, it's not universally applicable to all problems

Review: this Pointer

- this keyword in C++ is *an implicit pointer that points to the object of which the member function is called*
- Every object has its own this pointer. Every object can reference itself by this pointer
- Usage: resolve shadowing, access currently executing object

```
class Circle {  
public: // access specifier, introduced later  
    float x, y, r;  
  
    void setCenter(float x, float y) {  
        this->x = x;  
        this->y = y;  
    }  
  
    void setRadius(float r) {  
        this->r = r;  
    }  
};
```

Review: Constructor

- A constructor is a special member function that **initializes** member variables
- A constructor is **automatically called** when an object of that class is declared
- Rule I: a constructor must have the **same name as the class**
- Rule II: a constructor definition **cannot return a value**

Review: Constructor: Example-I

```
class Circle {  
public: // access specifier, introduced later  
    float x, y, r;  
  
    Circle() {  
        cout << "Input center:\n";  
        cin >> x >> y;  
        cout << "Input radius:\n";  
        cin >> r;  
    }  
};
```

```
int main() {  
    Circle *a = new Circle();  
    delete a;  
  
    Circle b; // Circle() will be called  
  
    return 0;  
}
```

Review: Constructor: Example-II

```
class Circle {  
public: // access specifier, introduced later  
    float x, y, r;  
  
    Circle(float x0, float y0, float r0) {  
        x = x0; y = y0; r = r0;  
    }  
};
```

```
int main() {  
    Circle a(0, 0, 1);  
  
    Circle *b = new Circle(1, 1, 2);  
    delete b;  
  
    // Note: A constructor cannot be called in the same  
    // way as an ordinary member function is called  
    a.Circle(1, 1, 1); // illegal  
  
    return 0;  
}
```

Review: Constructor: Example-III

- Constructor is typically overloaded, which allows objects to be initialized in multiple ways

```
class Circle {  
public: // access specifier, introduced later  
    float x, y, r;  
    Circle() {  
        cout << "Input center and radius:\n";  
        cin >> x >> y >> r;  
    }  
    Circle(float x0, float y0, float r0) {  
        x = x0; y = y0; r = r0;  
    }  
};
```

```
int main() {  
    Circle *a = new Circle();  
    delete a;  
  
    Circle b(0, 0, 1);  
  
    Circle c; // Circle() will be called  
    // A constructor behaves like a function that  
    // returns an object of its class type  
    c = Circle(1, 1, 2);  
  
    return 0;  
}
```

Review: Default Constructor

- The constructor with no parameters is the default constructor
- A default constructor will be generated by compiler automatically if NO constructor is defined

```
class Circle {  
public: // access specifier, introduced later  
    float x, y, r;  
    void setCenter() {  
        cout << "Input center:\n";  
        cin >> x >> y;  
    }  
    void setRadius() {  
        cout << "Input radius:\n";  
        cin >> r;  
    }  
};
```

```
int main() {  
    Circle a; // although no constructor is defined,  
              // the compiler will add an empty Circle()  
              // automatically, and call it when a  
              // Circle object is allocated  
  
    a.setCenter();  
    a.setRadius();  
  
    return 0;  
}
```

Review: Initializer List

- The list of members to be initialized is indicated with constructor as a **comma-separated** list followed by a **colon**.

```
class Circle {  
public: // access specifier, introduced later  
    float x, y, r;  
  
    Circle(int x, int y, int r):x(x), y(y), r(r) {}  
  
    // while is equivalent to  
    // Circle(int x0, int y0, int r0) {  
    //     x = x0; y = y0; r = r0;  
    // }  
};
```

Review: Initializer List

- **const** and **reference** member variables MUST be initialized using initializer list

```
class myClass {  
public: // access specifier, introduced later  
    const int t1;  
    int& t2;  
  
    // Initializer list must be used  
    myClass(int t1, int& t2):t1(t1), t2(t2) {}  
  
    int getT1() { return t1; }  
    int getT2() { return t2; }  
};
```

```
int main() {  
    int myint = 34;  
  
    myClass c(10, myint);  
  
    cout << c.getT1() << endl;  
    cout << c.getT2() << endl;  
  
    return 0;  
}
```

Review: Destructor

- A destructor is a special member function which is invoked automatically whenever an object is going to be destroyed
- Rule-I: a destructor has the same name as their class name preceded by a tiled (~) symbol
- Rule-II: a destructor has no return values and parameters
 - destructor overload is NOT allowed
- Statically allocated objects are destructed when the object is out-of-scope
- Dynamically allocated objects are destructed only when you delete them

Review: Destructor: Example

```
class Robot {
public: // access specifier, introduced later
    char *name = NULL;
    Robot(char *name) {
        int n = strlen(name);
        this->name = new char[n+1];
        strncpy(this->name, name, n);
        this->name[n] = '\0';
        cout << "Constructing " << name << endl;
    }
    ~Robot() {
        cout << "Destructing " << name << endl;
        // it's a good practice to free memories allocated
        // for member variables in destructor
        delete name;
    }
};
```

```
void func() {
    Robot eve("Eve");
    cout << "func is about to return\n";
    // Automatically calls the destructor when a
    // statically allocated object is out of the
    // scope
}

int main() {
    Robot *wall_e = new Robot("Wall-e");
    func();
    // A dynamically allocated object is destructed
    // only when you explicitly delete it
    delete wall_e;
    cout << "main is about to return\n";
    return 0;
}
```


Exercise 1

- Which statement(s) below is wrong?

```
struct CS2311Student {  
    int    sid;  
    float  quiz;  
    float  asg1;  
    float  asg2;  
};
```

```
int main() {  
    CS2311Student Jimmy;  
    cin >> Jimmy.sid; // statement a  
    CS2311Student Danny = {"123456", 80.0, 10.0, 10.0}; //statement b  
    Jimmy = Danny; // statement c  
    CS2311Student *dPtr = &Jimmy; // statement d  
    cout << dPtr.sid << endl; // statement e  
  
    return 0;  
}
```

Exercise 2

- Completing the definition of class Circle, so that the output of the program will be:

```
default (x y r) are: 0 0 0
Destructing 0 0 0
(x y r) are: 1 1 12
Destructing 1 1 12
```

```
class Circle {
private:
    float x, y, r;
public:

    ... // insert your codes here
};

int main() {
    Circle *a = new Circle();
    delete a;
    float r_ = 12;
    Circle b(1, 2, r_);
    return 0;
}
```

Exercise 2

```
int main() {  
    Circle *a = new Circle();  
    delete a;  
    float r_ = 12;  
    Circle b(1, 2, r_);  
    return 0;  
}
```

```
default (x y r) are: 0 0 0  
Destructing 0 0 0  
(x y r) are: 1 1 12  
Destructing 1 1 12
```

```
class Circle {  
private:  
    float x, y, r;  
public:  
    Circle():x(0.0), y(0.0), r(0.0) {  
        cout << "default (x y r) are: " << x << " " << y << " " << r << endl;  
    }  
    Circle(float x0, float y0, float r0):x(x0), y(y0), r(r0) {  
        cout << "(x y r) are: " << x << " " << y << " " << r << endl;  
    }  
    ~ Circle() {  
        cout << "Destructing " << x << " " << y << " " << r << endl; }  
};
```

Outline

- Access specifier: public, protect, and private
- Inheritance
- Operator overloading
- Polymorphism

Access Specifier

- An access specifier defines how the members (data fields and methods) of a class can be accessed
- **public**: members are accessible from outside the class
- **private**: members cannot be accessed from outside the class
- **protected**: members cannot be accessed from outside the class.

However, they can be accessed in inherited classes (later)

- By default, member variables and functions are private if no access specifiers are provided

Access Specifier: Example

```
class Actress {  
private:  
    int age; // error if access from outside  
  
public:  
    char name[255];  
    Actress(char *name, int age):age(age) {  
        strcpy(this->name, name);  
    }  
};
```

```
int main() {  
    Actress actressObj("Rachael", 25);  
  
    cout << actressObj.name << endl; // allowed  
    cout << actressObj.age << endl; // NOT allowed  
  
    // this is legal but ill-logical  
    // the name of an actress object should NOT  
    // be modified from outside  
    strcpy(actressObj.name, "Eve"); // allowed  
  
    return 0;  
}
```

Access Specifier (cont'd)

```
class Actress {  
private:  
    char name[255];  
    int age;  
  
public:  
    char name[255];  
    Actress(char *name, int age):age(age) {  
        strcpy(this->name, name);  
    }  
    char *getName() {  
        return name;  
    }  
};
```

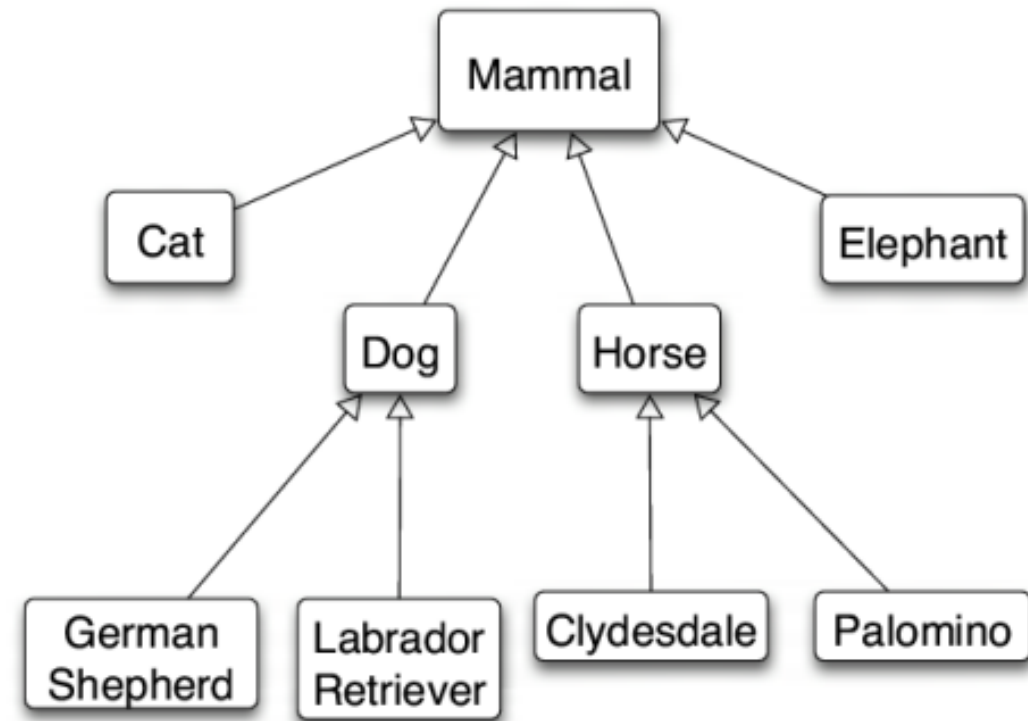
- We want actress name to be *read-only from outside*
- Declare name as private, and then define a public function to read it from outside

Access Specifier (cont'd)

- A common design of OOP is **data encapsulation**, which is to
 - define all member variables as private
 - provide enough get and set functions to read and write member variables
 - only functions that need to interact with the outside can be made public
 - supporting functions used by the member functions should also be made private

What is Inheritance

- **is-a relationship**: A hierarchical connection where one category can be treated as a specialized version of another.
 - every rectangle *is a* shape
 - every lion *is an* animal
 - every lawyer *is an* employee
- **class hierarchy**: A set of data types connected by *is-a* relationships that **can share common code**.
 - **Re-use**

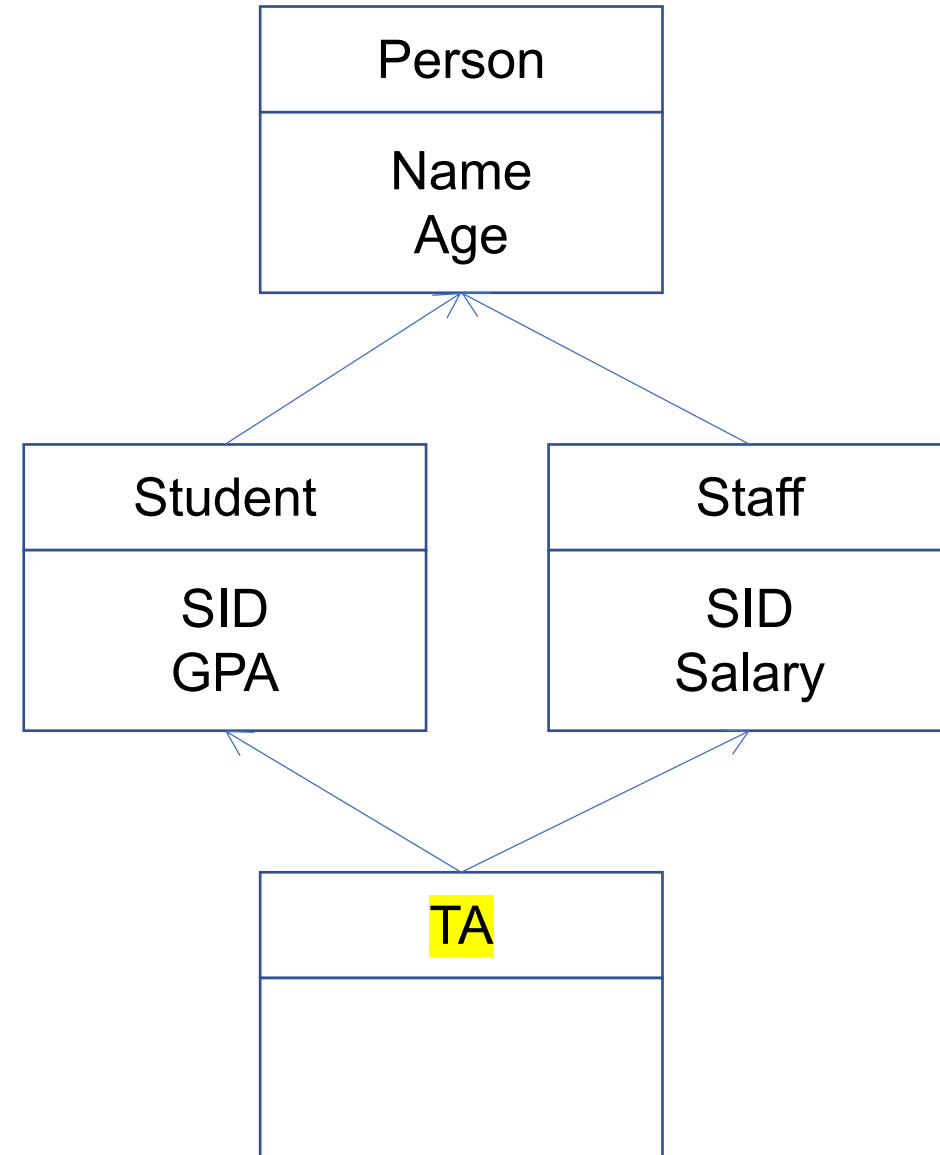


Basic Concepts

- **Inheritance**: A way to create new classes by extending existing classes
- **Base class**: Parent class that is being extended
- **Derived class**: Child class that inherits from base class(es)
 - A derived class gets a copy of every fields and methods from base class(es).
 - ❑ **Note**: gets a copy does NOT mean can access (details later)
 - A derived class can add its own behavior, and/or change inherited behavior

Basic Concepts

- Multiple inheritance: When one derived class has multiple base classes
- Forbidden in many object-oriented languages (e.g. Java) but allowed in C++.
- Convenient because it allows code sharing from multiple sources.
- Can be confusing or buggy, e.g. when both base classes define a member with the same name.



Syntax

```
class Parent { ... };
```

```
class Child : AccessSpecifier Parent { ... };
```

```
class ParentA { ... };
```

```
class ParentB { ... };
```

```
class Child : AccessSpecifier ParentA, AccessSpecifier ParentB { ... };
```

For example:

```
class TA : public Student, public Staff { ... };
```

Inheritance and Access

How inherited base class members appear in derived class

Base class members

```
class Parent {  
    private:  x;  
    protected: y;  
    public:   z;  
};
```

class Child : **public** Parent {...}

```
class Child {  
    // x is inaccessible  
    protected: y;  
    public:     z;  
};
```

class Child : **protected** Parent {...}

```
class Child {  
    // x is inaccessible  
    protected: y;  
    protected: z;  
};
```

class Child : **private** Parent {...}

```
class Child {  
    // x is inaccessible  
    private:   y;  
    private:   z;  
};
```

Public Inheritance: Example

```
class A {  
private:  
    int x;  
protected:  
    int y;  
public:  
    int z;  
};
```

```
class B : public A {  
public:  
    void print() {  
        cout << z; // allowed  
        y = 0; // allowed  
        cout << x; // NOT allowed  
    }  
};  
  
int main() {  
    B obj;  
    obj.y = 0; // NOT allowed, y is protected in B  
    obj.z = 0; // allowed, z is public in B  
    obj.print(); // allowed, print is public in B  
    return 0;  
}
```

Protected Inheritance: Example

```
class A {  
private:  
    int x;  
protected:  
    int y;  
public:  
    int z;  
};
```

```
class B : protected A {  
public:  
    void print() {  
        cout << z; // allowed  
        y = 0; // allowed  
        cout << x; // NOT allowed  
    }  
};  
int main() {  
    B obj;  
    obj.y = 0; // NOT allowed, y is protected in B  
    obj.z = 0; // NOT allowed, z is protected in B  
    obj.print(); // allowed, print is public in B  
    return 0;  
}
```

Private Inheritance: Example

```
class A {  
private:  
    int x;  
protected:  
    int y;  
public:  
    int z;  
};
```

```
class B : private A {  
public:  
    void print() {  
        cout << z; // allowed  
        y = 0; // allowed  
        cout << x; // NOT allowed  
    }  
};  
int main() {  
    B obj;  
    obj.y = 0; // NOT allowed, y is private in B  
    obj.z = 0; // NOT allowed, z is private in B  
    obj.print(); // allowed, print is public in B  
    return 0;  
}
```


Constructors in Inheritance

- Derived classes can have their own constructors
- When an object of a derived class is created, the base class's *default constructor* is executed first *at the beginning* of derived class's constructor, followed by executing the body of the derived class's constructor

```
class A {  
public:  
    A() { cout << "A's default constructor\n"; }  
};  
class B : public A {  
public:  
    B() {  
        cout << "B's constructor\n";  
    }  
};  
int main() {  
    B b;  
}  
// output will be:  
// A's default constructor  
// B's constructor
```

Constructors in Inheritance: Exercise

```
class A {  
public:  
    A() { cout << "A's default constructor\n"; }  
    A(int a) {  
        cout << "A's non-default constructor\n";  
    }  
};  
class B : public A {  
public:  
    B() {  
        cout << "calling A(2311) in B()\n"; A(2311);  
        cout << "calling A() in B()\n";    A();  
        cout << "B's constructor\n";  
    }  
};  
int main() {  
    B b;  
}
```

A's default constructor
calling A(2311) in B()
A's non-default constructor
calling A() in B()
A's default constructor
B's constructor

Passing Arguments to Constructors

```
class Student {
protected:
    int sid;
public:
    Student(int sid=0) : sid(sid) {}
    int getSid() { return sid; }
};

class TA: public Student {
protected:
    int courseid;
public:
    TA(int courseid =0) : courseid(courseid) {}
    int getCourseid() { return courseid; }
};
```

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

*How to pass parameters
to **base** constructor?*

```
int main() {
    Student sarah(12345);
    cout << sarah.getSid() << endl;

    TA bob(2311);
    cout << bob.getSid() << ": ";
    cout << bob.getCourseid() << endl;

    return 0;
}

// output will be:
// 12345
// 0: 2311
```

Passing Arguments to Constructors

- To pass arguments from child constructor to parent constructor
 - augment the parameter list of child constructor to include parent constructor parameters, and
 - call parent constructor in initial list

```
class B: public A {  
public:  
    B(B constructor parameters + A constructor parameters) : A(A constructor's args), ... {  
        ...  
    }  
};
```

Passing Arguments to Constructors

```
class Student {
protected: int sid;
public:     Student(int sid=0) : sid(sid) {}
           int getSid() { return sid; }
};
class TA: public Student {
protected: int courseid;
public:     TA(int sid=0, int courseid=0) : Student(sid), courseid(courseid) {}
           int getCourseid() { return courseid; }
};
int main() {
    int sid=12345, courseid=2311;
    TA bob(sid, courseid);
    cout << bob.getSid() << ": " << bob.getCourseid() << endl;
    return 0;
}
```

Destructors in Inheritance

- Derived classes can have their own destructors
- When an object of a derived class is destroyed, the derived class's destructor is executed first, followed by the base class's destructor

```
class A {  
public:  
    ~A() { cout << "A's destructor\n"; }  
};  
class B : public A {  
public:  
    ~B() { cout << "B's destructor\n"; }  
};  
int main() {  
    B* b = new B();  
    delete b;  
    return 0;  
}  
// The output will be:  
// B's destructor  
// A's destructor
```

Outline

- Access specifier: public, protect, and private
- Inheritance
- Operator overloading
- Polymorphism

Operator Overloading

- Enabling C++'s operators to work with class objects
- Using traditional operators with **user-defined objects**
- Examples of already overloaded operators
 - Operator << is both the stream-insertion operator and the bitwise left-shift operator
 - + and -, perform arithmetic on multiple types

```
int a = 1;
int b = 2;
if (a < b)
    cout << "a < b " << endl;
```

```
class Triangle { ... };
Triangle a, b;
...
if (a < b) // errors without overloading
    cout << "a < b " << endl;
```


Operator Overloading

`+` `-` `*` `/` `%` `^` `&` `|` `~` `!` `,` `=` `<` `>` `<=` `>=`

`++` `--` `<<` `>>` `==` `!=` `&&` `||` `+=` `-=` `*=` `/=`

`%=` `^=` `&=` `|=` `<<=` `>>=` `[]` `()` `->`

`->*` `new` `new[]` `delete` `delete[]`

Operator Overloading

```
class Triangle {  
private:  
    double s1, s2, s3;  
    double area;  
public:  
    Triangle() {}  
    void setSides();  
    void computeArea();  
    double getArea();  
};
```

```
Triangle a, b;
```

```
a.setSides();  
b.setSides();
```

**lhs = Left
Hand Side**

**rhs = Right
Hand Side**

```
if (a < b) {  
    cout << "Triangle a is smaller than triangle b\n";  
}
```

Operator Overloading

- Overloading an operator
 - Write function definition as normal
 - Function name is keyword **operator** followed by the symbol for the operator being overloaded
 - ❑ **operator+** used to overload the addition operator (+)
- Special operators
 - To use an operator on a class object it must be overloaded except the assignment operator(=) or the address operator(&)
 - ❑ Assignment operator by default performs member-wise assignment
 - ❑ Address operator (&) by default returns the address of an object

Operator Overloading: Member Function

- Add a function called operator _ (e.g., <, +, !) to your class:

```
class Triangle {  
private:  
    double s1, s2, s3;  
    double area;  
public:  
    Triangle() {}  
    void setSides(...);  
    double getArea(...);  
    bool operator< (Triangle & rhs);  
    // lhs (left hand side) of operator < is this.  
};  
bool Triangle::operator<(Triangle & rhs) {  
    if (area < rhs.area) return true;  
    else return false;  
}  
...
```

```
#include <iostream>  
using namespace std;  
  
int main() {  
    Triangle a; a.setSides(...);  
    Triangle b; b.setSides(...);  
    ...  
    cout << (a < b);  
    return 0;  
}  
  
// a < b  $\leftrightarrow$  a.operator<(b)
```

Operator Overloading: Friend Function

- *Friend function*: a special function which is a non-member function of a class but has privilege to access private and protected data of that class
- Friend function can be declared in any section of the class i.e. public or private or protected
- When friend function is called, neither name of object nor dot operator is used

Operator Overloading: Friend Function

```
class Triangle {  
private:  
    double s1, s2, s3;  
public:  
    Triangle() { s1=0; s2=0; s3=0; }  
    Triangle(double s1, double s2, double s3): s1(s1), s2(s2), s3(s3) {}  
    double getArea();  
    friend bool operator< (Triangle &lhs, Triangle &rhs);  
    friend bool operator> (Triangle &lhs, Triangle &rhs);  
    friend ostream& operator<< (ostream &outs, Triangle &c);  
};  
double Triangle::getArea() {  
    double s = (s1+s2+s3)/2;  
    return sqrt(s*(s-s1)*(s-s2)*(s-s3));  
}
```

output stream in C++

- *output stream* in C++: an abstract data type that represents a sequence of characters that can be sent to an output device, such as the console, a file, or a network connection
- To output text to an output stream, simply use the insertion operator (`<<`) to insert the text into the stream
 - ❑ `cout << "Hello World!" << endl;`
- can be used to output a wide range of data types, including integers, floating-point numbers, characters, strings, and user-defined types

Operator Overloading: Friend Function

```
bool operator<(Triangle &lhs, Triangle &rhs) {
    return lhs.getArea() < rhs.getArea();
}

bool operator>(Triangle &lhs, Triangle &rhs) {
    return lhs.getArea() > rhs.getArea();
}

ostream &operator << (ostream &outs, Triangle &t) {
    outs << "The sides are: ";
    outs << t.s1 << " " << t.s2 << " " << t.s3 << " ";
    outs << "The area is: ";
    outs << t.getArea() << endl;
    return outs;
}
```

```
int main() {
    Triangle t1(3, 4, 5);
    Triangle t2(5, 6, 7);
    cout << t1;
    cout << t2;
    if (t1 < t2) {
        cout << "t1 is smaller\n";
    } else {
        cout << "t2 is smaller\n";
    }
    return 0;
}
```


Outline

- Access specifier: public, protect, and private
- Inheritance
- Operator overloading
- Polymorphism

Type Casting in Class Inheritance

```
class Animal {  
    ...  
};  
class Human : public Animal {  
    ...  
};  
  
// Only down-casting is allowed in class type conversion  
// You can say a human is an animal, but not vice versa  
  
Animal *a = new Human();    // legal  
Human *b = new Animal();    // illegal
```

Static Type vs Dynamic Type

- *Static type*: the declared type
- *Dynamic type*: the actual type assigned

```
class Animal {  
    ...  
};  
  
class Human : public Animal {  
    ...  
};  
  
class Dog : public Animal {  
    ...  
};
```

```
int main() {  
    Human *human = new Human();  
    Dog *dog = new Dog();  
  
    Animal *a; // the static type of a is Animal  
    a = human; // the dynamic type of a is Human  
    a = dog;   // the dynamic type of a is Dog  
  
    delete human;  
    delete dog;  
    return 0;  
}
```

Override

- To re-implement a base class's member function by writing a **new version** of that function (with the same function prototype) in a **derived class**

```
class Shape {
public:
    void print() { cout << "I am a shape\n"; }
};
class Circle: public Shape {
private:
    double radius;
public:
    Circle(double radius_):radius(radius_) {};
    void print() { cout << "I am a circle and my radius is " << radius << "\n"; }
};
```

Override vs Overload

- Overload

```
double sum(double, double, double);  
double sum(double, double);
```

- Override

```
void Animal::makeSound();  
void Human::makeSound();  
void Dog::makeSound();
```

Polymorphism

- Polymorphism means "many forms"
- In inherited classes, the same function behaves differently depending on types

```
void Animal::makeSound();  
void Human::makeSound();  
void Duck::makeSound();  
void Dog::makeSound();  
void Cat::makeSound();
```



Polymorphism: Static Binding

- The called function is determined by *static type*

```
class Animal {  
    public:  
    void sayHi() {  
        cout << "...\\n";  
    }  
};
```

```
class Human : public Animal {  
    public:  
    void sayHi() {  
        cout << "hi\\n";  
    }  
};
```

```
class Dog : public Animal {  
    public:  
    void sayHi() {  
        cout << "wow wow\\n";  
    }  
};
```

```
int main() {  
    Human *human = new Human();  
    Dog *dog = new Dog();  
    Animal *a;  
  
    // the static type of a is Animal  
    a = human;  
    a->sayHi(); // will print "..."  
    a = dog;  
    a->sayHi(); // will print "..."  
  
    delete human;  
    delete dog;  
    return 0;  
}
```



Polymorphism: Dynamic Binding

- We want the called function to be determined by *dynamic type*

```
int main() {  
    Human *human = new Human();  
    Dog *dog = new Dog();  
  
    Animal *a;           // the static type of a is Animal  
    a = human;           // the dynamic type of a is Human  
    a->sayHi();           // we want it to print "Hi"  
    a = dog;             // the dynamic type of a is Dog  
    a->sayHi();           // we want it to print "wow wow"  
  
    delete human;  
    delete dog;  
    return 0;  
}
```


Dynamic Binding: Virtual Function

- A *virtual function* is declared in the base class using the keyword `virtual` and is re-defined (Overridden) in the derived class
- Virtual functions are Dynamic in nature
- Allows dynamic binding at runtime

Polymorphism: Dynamic Binding

```
class Base {  
public:  
    virtual void print() {  
        cout << "print base\n";  
    }  
    void show() {  
        cout << "show base\n";  
    }  
};
```

```
class Derived : public Base {  
public:  
    void print() {  
        cout << "print derived\n";  
    }  
    void show() {  
        cout << "show derived\n";  
    }  
};
```

```
int main() {  
    Base *base;  
    Derived *derived = new Derived();  
  
    base = derived;  
    base->print(); // dynamic binding  
                  // will print "print derived"  
  
    base->show(); // static binding  
                  // will print "show base"  
  
    delete derived;  
    return 0;  
}
```

Exercise

What is printed?

- (A) “Mammal”
- (B) “Cat”
- (C) “Siamese”
- (D) Gives an error
(identify compiler or crash)
- (E) “rawr”
- (F) “meow”
- (G) Other/none/more

```
class Mammal {
public:
    virtual void makeSound() = 0;
    void printString() { cout << "Mammal" << endl; }
};

class Cat : public Mammal {
public:
    virtual void makeSound() { cout << "rawr" << endl; }
    void printString() { cout << "cat" << endl; }
};

class Siamese : public Cat {
public:
    virtual void makeSound() { cout << "meow" << endl; }
    void printString() { cout << "Siamese" << endl; }
    virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};

Siamese * s = new Siamese();
s->printString();
```

Exercise

What is printed?

- (A) "Mammal"
- (B) "Cat"
- (C) "Siamese"
- (D) Gives an error
(identify compiler or crash)
- (E) "rawr"
- (F) "meow"
- (G) Other/none/more

```
class Mammal {
public:
    virtual void makeSound() = 0;
    void printString() { cout << "Mammal" << endl; }
};

class Cat : public Mammal {
public:
    virtual void makeSound() { cout << "rawr" << endl; }
    void printString() { cout << "cat" << endl; }
};

class Siamese : public Cat {
public:
    virtual void makeSound() { cout << "meow" << endl; }
    void printString() { cout << "Siamese" << endl; }
    virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};

Siamese * s = new Mammal();
s->printString();
```

Exercise

What is printed?

- (A) “Mammal”
- (B) “Cat”
- (C) “Siamese”
- (D) Gives an error
(identify compiler or crash)
- (E) “rawr”
- (F) “meow”
- (G) Other/none/more

```
class Mammal {
public:
    virtual void makeSound() = 0;
    void printString() { cout << "Mammal" << endl; }
};

class Cat : public Mammal {
public:
    virtual void makeSound() { cout << "rawr" << endl; }
    void printString() { cout << "cat" << endl; }
};

class Siamese : public Cat {
public:
    virtual void makeSound() { cout << "meow" << endl; }
    void printString() { cout << "Siamese" << endl; }
    virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};

Mammal * s = new Siamese();
s->printString();
```

Exercise

What is printed?

- (A) “Mammal”
- (B) “Cat”
- (C) “Siamese”
- (D) Gives an error
(identify compiler or crash)
- (E) “rawr”
- (F) “meow”
- (G) Other/none/more

```
class Mammal {
public:
    virtual void makeSound() {cout << " " << endl;}
    void printString() { cout << "Mammal" << endl; }
};

class Cat : public Mammal {
public:
    virtual void makeSound() { cout << "rawr" << endl; }
    void printString() { cout << "cat" << endl; }
};

class Siamese : public Cat {
public:
    virtual void makeSound() { cout << "meow" << endl; }
    void printString() { cout << "Siamese" << endl; }
    virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};

Mammal * s = new Mammal();
s->printString();
```

Exercise

What is printed?

- (A) “Mammal”
- (B) “Cat”
- (C) “Siamese”
- (D) Gives an error
(identify compiler or crash)
- (E) “rawr”
- (F) “meow”
- (G) Other/none/more

```
class Mammal {
public:
    virtual void makeSound() {cout << " " << endl;}
    void printString() { cout << "Mammal" << endl; }
};

class Cat : public Mammal {
public:
    virtual void makeSound() { cout << "rawr" << endl; }
    void printString() { cout << "cat" << endl; }
};

class Siamese : public Cat {
public:
    virtual void makeSound() { cout << "meow" << endl; }
    void printString() { cout << "Siamese" << endl; }
    virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};

Mammal * s = new Cat();
s->makeSound();
```

Labs/Assignments

- Pay attention to the dues

Outline for Lec1 Introduction

- What's a computer
 - Numbering system
 - Logic gates and circuits
 - Stored Program Computer and ISA
- Programming languages
 - Machine/ Symbolic/High-level Language
 - Compiler
- Basics of computer programming
 - Logic Flow
 - Developing: Coding / Compilation/ Linking
 - C++ Syntax: Function/Statement/...

Outline for Lec2 Data, Operators, and BasicIO

- C++ basic syntax
- Variable and constant
- Operators
- Basic I/O

Outline for Lec3 Control Flow - Condition

- Logical data type, operators and expressions
- If statement
 - Simple
 - Nested
- Switch statement

Outline for Lec4 Control Flow - Loop

- Loop
 - while
 - do-while
 - for
- Programming styles for control flow

Outline for Lec5 Function

- Defining a function
- Calling a function
- Declare a function (function prototype)
- Passing parameters
- Recursive functions

Outline for Lec6 Array

- Array definition
- Array initialization
- Passing array to functions
- Array operations
- Multi-dimensional array

Outline for Lec7 String

- *char* recap
- C string basics
- Reading and printing C strings
- Common string functions
- Safety of string functions
- File I/O (Lec8)

Outline for Lec9 Pointer I

- Recap: variable and memory
- Pointer and its operations
- Pass by pointer
- Array and pointer

Outline for Lec10 Pointer II

- Pointer arithmetic
- Pointer array vs Array pointer
- Pointer of pointer & Pointer reference
- Dynamic memory allocation

Outline for Lec11 & 12

- Prerequisite: C-like struct and overload
- Class and objects: basic concepts and syntax
- Constructors and destructors
- Access specifier: public, protect, and private
- Inheritance
- Operator overloading
- Polymorphism

As for Final

- Please remember to check **when and where!!!**
- It consists of **6** questions in **2 sections**. Answer ALL questions in Section A and TWO questions in Section B.
 - Write down outputs
 - Correct the errors
 - Complete the programs
 - Implement a design

Sample Question 1

- Write the console output of the following programs. If there is a compilation error, write "Compilation error", and then briefly explain the cause.

```
#include <iostream>
using namespace std;
int main() {
    int x = -1, y = 0;
    if (x)
        if (y)
            cout << "A";
    else
        cout << "B";
    cout << "C";
    return 0;
}
```

Sample Question 2

- Write the console output of the following programs. If there is a compilation error, write "Compilation error", and then briefly explain the cause.

```
#include <iostream>
using namespace std;
class A {
public:
    char name = 'A';
};
class B : private A {
public:
    B() { name++; }
};
int main() {
    B* b = new B();
    cout << b->name;
    delete b;
    return 0;
}
```

Sample Question 3

- Define a class for a type called Fraction. This class is used to represent a ratio of two integers.
- It should
 - Include functions that allow the user to set the numerator and the denominator.
 - Include a member function that returns the value of the numerator divided by the denominator as a double.
 - Include an additional member function that outputs the value of the fraction reduced to lowest terms. For example, instead of outputting 20/60 the function should output 1/3.