Comp2014 Object Oriented Programming

Lecture 10

Operator Overloading, References and Friends

Topics covered in last lecture

- Function calls and binding
- Static binding
- Function overriding and dynamic binding
- Polymorphism
- Virtual functions
- Abstract classes

Difficulty factor:
*****

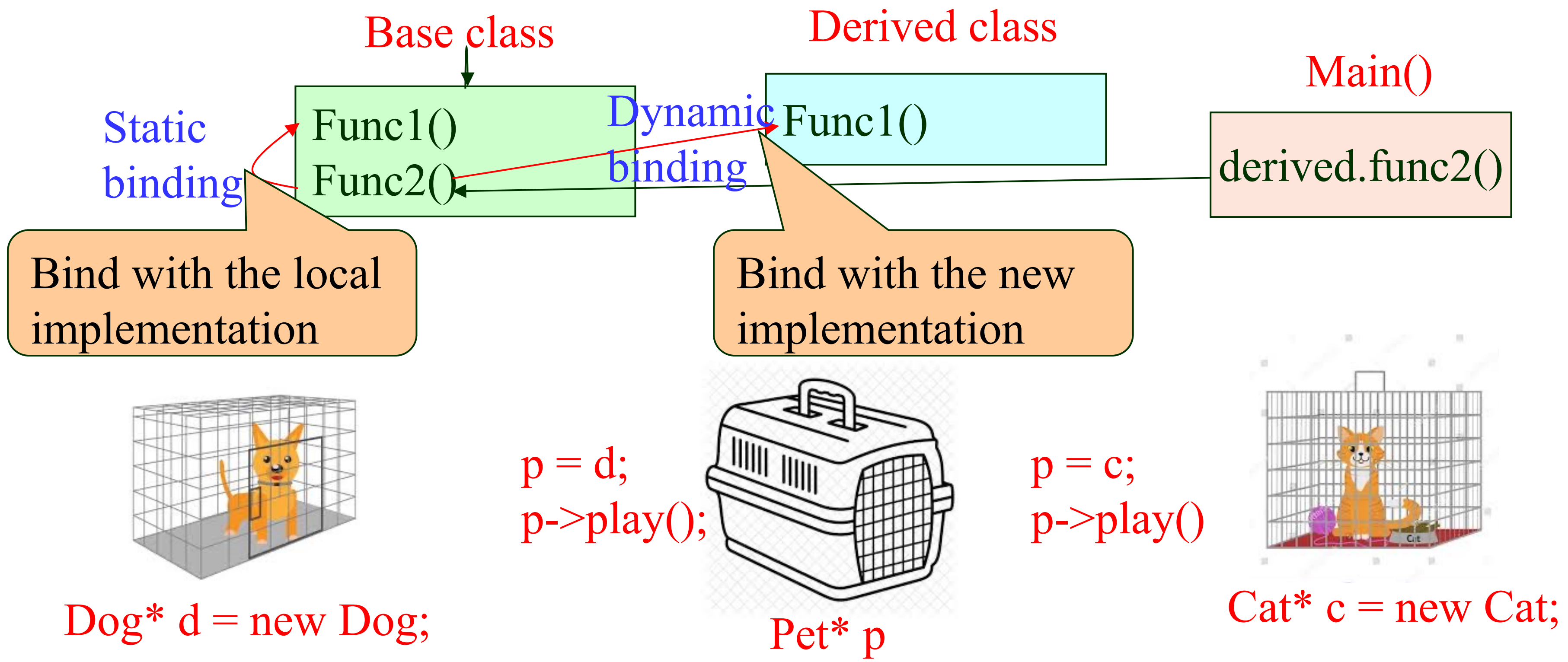
```
//function prototype in a class
Type className::functionName(Type1 parameter1, Type2 parameter2)

//function call
className object;
Type o = object.functionName(object1, object2);
```

binding

Polymorphism

- The way that C++ determines which function to call is through two types of binding, *static* and *dynamic*.
- *Polymorphism* permits the same function name to invoke one response in objects of a base class and another response in objects of a derived class.



```
Dog* d = new Dog;
Pet* p
p = d;
p->play();
p = c;
p->play()
Cat* c = new Cat;
```

Topics covered by the lecture

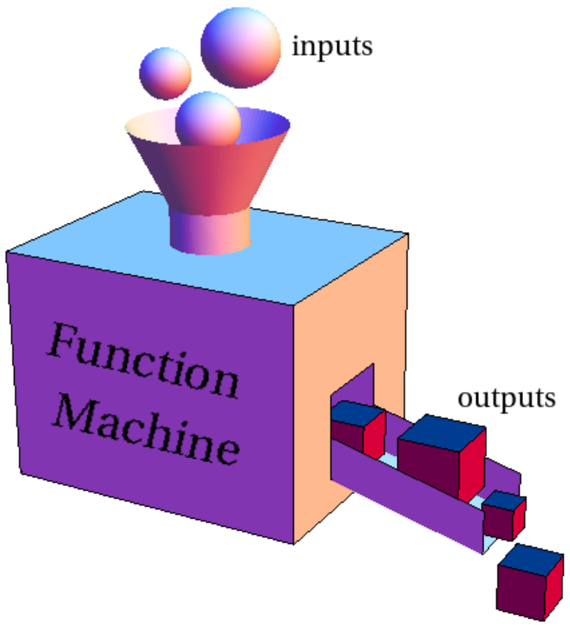
- Function and function overloading (review)
- Operator overloading
- References
- Friends
- Review of assignment 1 and preview of assignment 2

Difficulty factor:
****

Functions and operators

- A function is a process that takes inputs and generates outputs.
- A function can be overloaded, e.g.,

```
int addition(int x, int y)
double addition(double x, double y)
```



```
int addition(int x, int y) {
  int z = x + y;
  return z;
}

int main() {
  cout<< addition(20,30) <<endl;
return 0;
}
```

Built-in operators

- Beside the functions we created, many mathematical operators are also functions and defined for built-in data types, such as

  +, -, *, /, %: arithmetic operations (for *int, float, double, long*)

  <=, <, >=, >, ==, !=: relational operations (for *expressions*)

  &&, ||: logical operations (for *Boolean variables*)

  These *built-in* functions are called *operators.* For instance,

```
int value, x, y, z;
z = x + y;
value = (2*x + y)/z;
x <= y;
bool u, v, w;
u && v;
w == u || v;
```

Built-in operators

- Other built-in operators you may be less aware of:
  - Incremental operators: ++, --
  - Accumulation operators: +=, -=, *=, /=, ^=
  - Stream operators (free functions): <<, >>
  - Array operators: [ ]

```
int value, x, y, z;
ifstream fin;
ofstream fout;
value += x; //means value = value + x;
y = ++X; //means x=x+1; and y=x;
fin >> x;  fout << y;
array[x] //  the xth value of array
```

Functions and operators

- An operator can be either *prefix*, *infix* or *postfix:*

  Prefix: --x, ++x, &value

  Postfix: x++, y--

  *Infix: x + y, x*y, x < y , x == y, u && v*

Unless prefix, infix and postfix operators are most for human users. The implementation of all operators can be represented as function in prefix format:

--x	*x.operator--()*	*x--*	*operator--(x)*
x + y	*x.operator+(y)*		
*x * y*	*x.operator*(y)*		
x < y	*x.operator<(y)*		
u && v	*u.operator&&(v)*		

Built-in operators (i.e. functions)

- Are these built-in operators applicable to the user-defined data-types, including the classes you created? NO

```
class Date {
public:
    int day;
    int month;
    int year;
};
```

```
int main() {
    Date d(30,9,2021), d1;
    d1 = d + 100;
    while (d < d1)
        d++;
}
```

Next day

- If we want the operators of +, < or ++ to be used for Date data type, we need them to be overloaded to that data type, i.e,

```
Date operator+(int), Date operator<(Date),
 Date operator++()
```

Operator Overloading

Built-in operators can be overloaded so that when they are used with class objects, the operators have meaning appropriate to the new types. In fact, to use an operator on non built-in data type, the operator *must* be overloaded in the class, with two exceptions:

1). *The assignment operator,* =, *will allow copying of contents of corresponding data members called member-wise assignment. This operation, though, is dangerous if any of the data members are pointers (shallow copy)*

2). *The other exception is the address–of operator,* &. *It returns the address of the object in memory for any data type.*

Operators that can be overloaded

- Almost all operator can be overloaded but common ones are:
  - Arithmetic operators: + - * / %
  - Incremental operators: ++ --
  - Comparison operators: != == > < >= <=
  - Boolean operators: && ||
  - Accumulation operators: += -= *= /= ^=
  - Stream operators (free functions): << >>
  - Array operators: [ ]
  - Assignment and address of: =

Commonly for search and comparison

Commonly for easy i/o operations

Commonly for deep copy

Unary operators: --x, ++x, …

Binary operators: x + y, x || y, cout >> x, …

Overload an operator only if it is necessary or actually helpful!

Operator Overloading - Syntax

- To overload an operator "ⓘ" in class className,
  - If "ⓘ" is a unary operator, define a public function:

    className operatorⓘ();
  - If "ⓘ" is a binary operators, define a public unary function:

    className operatorⓘ(className x);
  - Note that this rule may vary for special cases!
- Examples:

```
x++        Overloading: yourType operator++()
x = y ;    Overloading: yourType operator=(yourType)
```

Where is the other argument?

Example of operator overloading

- Overload < for class Date

```
class Date {
  private:
     int day;
     int month;
     int year;
  public:
    Date(int,string,int);
    void display();
    bool operator<(Date d);
};
```

```
bool operator<(Date d) {
    if(year < d.year )
          return true;

    if(year == d.year && month < d.month)
          return true;

    if(year == d.year && month == d.month
&& day < d.day)
          return true;

    return false;
}
```

dateOverloading.cpp

Overloading assignment operator =

Overload `operator=` for class `DynamicArray`

```
class DArray {
private:
  int* arr;
  int size;
public:
  DArray(int s):size(s) {
     arr = new int[s];
  }

  ~DArray() {
     delete[] arr;
  }
};
```

- Shallow copy:

```
void operator= (DArray c) {
        this->size = c.size;
        arr = c.arr;
}
```

- Deep copy:

```
void operator=(DArray c) {
        this->size = c.size;
        arr = new int[size];
        for(int i=0;i<size;i++)
          arr[i] = c.arr[i];
}
```

Topics covered by the lecture

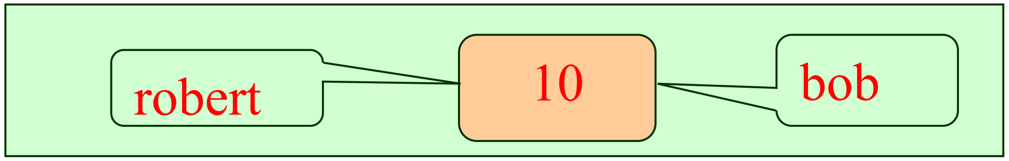
- Function and function overloading (review)
- Operator overloading
- References
- Friends
- Review of assignment 1 and preview of assignment 2

References

Why does a variable need another name?

- **Reference**: a reference is an *alias of an object variable*, that is, another name for an already existing variable.

```
int robert;
int& bob = robert;
```



```
robert = 10; //bob is 10 now
bob = 20; // robert equals 20 as well
```

  – *bob* is a reference to the storage location of *robert*

  – Changes made to *bob* will apply to *robert,* so does *bob.*

- Benefit: Two variables share the same memory cell.

- Call-by-reference is an example of using references

reference.cpp

Reference

Mind deference with Java

Things you should always remember about references.

- A reference must always refer to something. NULL is not allowed.
- A reference must be initialized when it is created. An unassigned reference can not exist.
- Once initialized, it cannot be changed to another variable.

```
int main() {
  int a=9;
  int& aref = a;
  a++;
  cout << "The value of a is " << aref;
  return 0;
}
```

Date.Java

Reference vs pointer

- References are often confused with pointers but there are three major differences between them
  - You cannot have a NULL reference, but you can assign NULL to a pointer.
  - Once a reference is initialized to a variable, it cannot be referred to another variable. Pointers can point to any objects at any time if type matches.
  - A reference must be initialized when it is created. Pointers can be initialized at any time.

A reference is a kind of constant pointer.

DateReference

Friends

- Encapsulation is an OOP concept that binds together the data and functions, and that keeps both safe from outside interference and misuse.
- Normally a class restricts outside accesses to its data and hence these data are not accessible to data members or member functions of other classes and free functions.
- There may be some circumstances where the programmer wishes to allow such accesses. << and >> operators are typical examples.
- This can be done via the use of **friends**.
- Friends can be either functions or classes

Friend Functions of Classes

- Friend function (of a class): a non-member function of the class that has access to all the members of the class, including private members
- Declare a **`friend`** function in a class

```
class classIllusFriend {
    friend void homestay(/*parameters*/);
    …
};
```

Function homestay() is not a member function of classIllusFriend but can access all the data members and member functions of this class

- Friend class is similar

Friend.cpp   date.h

Topics covered by the lecture

- Function and function overloading (review)
- Operator overloading
- References
- Friends
- Review of assignment 1 and preview of assignment 2

Homework

- Read textbook Chapter 8.
- Online tutorial 3 will be open next week.
- Practical 7 will be checked from this week.
- Assignment 2 will be due next Friday 5 pm 14 Oct 2022.