Object-Oriented Programming: More on Class

Lectured by Ming-Te Chi 紀明德

First Semester, 2022

Computer Science Department
National Chengchi University

Slides credited from 李蔡彥 and 廖峻鋒

More on Class

- More on I/O classes
- `friend` classes and functions
- `this` pointer
- Static data and member functions
- Convert constructors
- Copy constructors

More On I/O: Buffered and Unbuffered I/O

- `cout` is buffered.

```
void main() {
    cout << "Hello" << '\n';
    while(1) {
        int x=1;
    }
}
```

"Hello" is not printed until you kill the program.

- When is `cout` flushed?
  - An input statement
  - End of the program
  - A specified flush command
- `cerr`: unbuffered `cout`
- `clog`: buffered `cerr`
- `cin`: buffered until you hit return

```
cout << "Hello" << endl;
cout << "Hello" << flush;
```

'endl' is the same as '\n' except that it causes an immediate flush.

More On I/O: `cin` Member Functions

- `cin` is an object. One of the member functions is `get()`, which exists in three overloaded forms.
- Prototype #1

  ```
  istream &get(char &destination);
  ```

```
char c, d, e;
cin >> c;        // skip white space
cin.get(c);      // get() returns white space
cin.get(c).get(d).get(e);
```

cin Member Functions (II)

- Prototype #2
  `istream &get(char *buffer, int length, char delimiter='\n');`
  - Read up to length-1 or the delimiter character.
  - The buffer is automatically terminated with null.

```
char buffer1[100], dummy, buffer2[100];
cin.get(buffer1, 100);
cin.get(dummy);
cin.get(buffer2, 100);
```

Caution: get() does not remove the delimiter character from the stream.

- Prototype #3
  `int get();`
  - The purpose of this function is to be able to return `EOF`. Note it returns `int`.

`cin` Member Functions (III)

- `getline()`

  ```
  istream &getline(char *buffer, int length, char delimiter='\n');
  ```

  - Like the prototype #2 of get except that it eats the delimiter.

- `ignore()`

  ```
  istream &ignore(int length=1, int delimiter=eof);
  ```

  - ignore() skips over length characters or until the delimiter is reached.

- `peek()`

  ```
  int peek();
  ```

  - Returns the next character without removing it.

- `putback()`

  ```
  istream &putback(char c);
  ```

  - Put a character back to the stream

More On I/O: Testing the State of the Stream

- You can test the state bits of `ios` through its member functions.

```
int getSum() { // compute the sum of numbers you input
    char badData;
    int number, sum=0;
    while(true) {
        cout << "Type a number:";
        cin >> number;
        if (cin.good()) {       // correct input
            if (number==0)      // input 0 to quit
                return sum;
            sum += number;
        } else if (cin.fail()) {// error in input, continue
            cin.clear();            // reset state bits
            cin.get(badData);       // skip the bad data
            cout << badData << "is not a number\n";
        } else if (cin.bad()) { // stream corrupted
            return sum;
        }
    }
}
```

Basic File I/O Operations

- A ‘cat’ program in Unix: implicit open

```
#include <fstream>
#include <iostream>
using namespace std;
int main() {
    char letter;
    ifstream myFile("test.dat"); // implicit open
    if (!myFile) { // if the failbit or badbit is set
        cerr << "Cannot open file: test.dat";
        exit;
    }
    while(myFile.get(letter)) { // get return false on EOF
        cout << letter;
    }
}
```

- Explicit open:

```
ifstream myFile; // do not open yet, can be reused
myFile.open("test.dat"); // explicit open
myFile.close(); // close
```

File I/O With << and >> Operators

- You can use the << and >> operator in the same way as for the console objects.

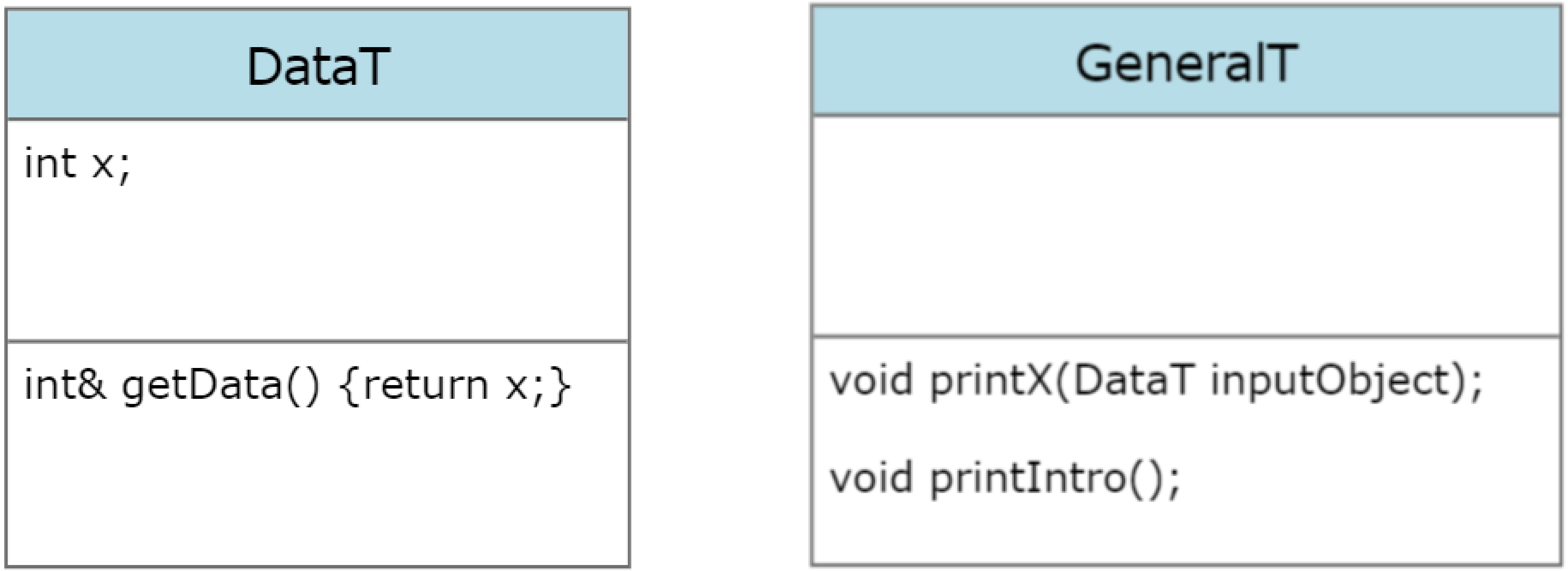
```
#include <fstream>
#include <iostream>
using namespace std;
int main() {
    int n1=10, n2=20, n3=30, number;
    ofstream outFile("test.dat"); // open output file
    if (!outFile) {
        cerr << "Cannot open file: test.dat";
        exit;
    }
    outFile << n1 << ' ' << n2 << ' ' <<  n3; // ' ' needed

    ifstream inFile("test.dat"); // open input file
    if (!inFile) {
        cerr << "Cannot open file: test.dat";
        exit;
    }
    while(inFile >> number) // get return false on EOF
        cout << number;     // white spaces are skipped
}
```

Output:
10 20 30

Class

friend, this, static, copy constructor


DataT
int x;
int& getData() {return x;}
GeneralT
void printX(DataT inputObject);
void printIntro();

Friends: Granting Friendship

- What is `friend`? Another class/function that can access your private data. Why do we need friends?

```
class DataT;
class GeneralT {
  public:
    void printX(DataT inputObject); // needs friendship
    void printIntro();
};

class DataT {
  friend GeneralT; // give GeneralT access to private data
  public:
    DataT(int x);
  private:
    int x;
  private: // need to be public if friendship is not granted
    int& getData() {return x;}};
};
void GeneralT::printX(DataT inputObject) {
    cout << inputObject.getData() << "\n";
    cout << inputObject.x << "\n";
}
```

`friend GeneralT::printX(DataT inputObject);`
will grant friendship to the printX function only

Details about Friends

- What does the `friend` declaration go?
  - Anywhere; the access specifiers have no meaning.
  - Most commonly on the top of the class declaration.

```
class DataT {
  friend GeneralT;
public:
    DataT(int x); …
}
```

- Friend classes (functions) cannot access the other class directly. It needs an object of the other class.

illegal

```
void GeneralT::printX() {
    cout << getData();
}
```

legal

```
void GeneralT::printX(DataT inputObject) {
    cout << inputObject.getData();
}
```

- Friendship is granted, not taken.
  - If class A grants friendship to class B, that does not make class A a friend of class B.

A Linked List Implementation with Friends

- Two classes: one for the data and the other for the list.

```
class DataT {
  friend class LinkedListT;
  private:
    DataT(int inValue);
    int value;
    DataT *next;
};
class LinkedListT {
  public:
    LinkedListT();
    ~LinkedListT();
    void append(int inValue);
    void display();
  private:
    DataT *tail;
    DataT *head;
};
```

Note:

- **DataT has no public interface. Even the constructor is private.**
- **The client will never create a DataT object. DataT exists only as an auxiliary class to LinkedListT.**
- **Advantages of using friend here:**
  - Avoid having public interface in DataT.
  - Efficiency.

A Linked List Implementation (Cont.)

```
// constructor for DatatT
DataT::DataT(int inValue): value(inValue), next(nullptr){
}
// constructor for LinkedListT
LinkedListT::LinkedListT(): head(nullptr), tail(nullptr) {
}
// append function in LinkedListT
void LinkedListT::append(int value) {
    DataT *temp=new DataT(value);
    if (head== nullptr) {
        head = temp;
        tail = temp;
    } else {
        tail->next=temp;
        tail=temp;
    }
}
// display function in LinkedListT
void LinkedListT::display() {
    DataT *cur;
    for(cur=head; cur!= nullptr; cur = cur->next)
        cout << cur->value << " ";
}
```

```
int main() {
    LinkedListT myLinkedList;
    myLinkedList.append(1);
    myLinkedList.append(2);
    myLinkedList.append(3);
    myLinkedList.display();
}
```

Output:
1 2 3

The Pointer **this**

- `this` is a pointer to the object itself.

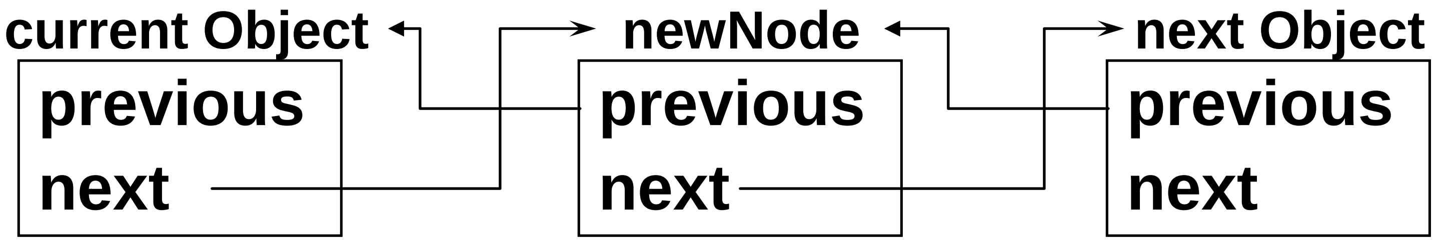
```
class PointT {
  public:
    PointT();
    int add() const;
    void setValues(int x, int y);
    bool equal(PointT other) const;
  private:
    int x, y;
};
int PointT::add() const {
    return x + y;   // the same as return this->x + this->y;
}
void PointT::setValues(int x, int y) {
    this->x = x;    // local variables take precedence
    this->y = y;
}
bool PointT::equal(PointT other) const {
    return (x == other.x) && (y == other.y);
}
```

Why Is **`this`** Good For?

- `this` is most commonly used to link to other objects.

```
class LinkedListT {
  public:
    void insert(LinkedListT *newNode);
  private:
    LinkedListT *previous;
    LinkedListT *next;
};

void LinkedListT::insert(LinkedListT *newNode) {
    newNode->next = next; // currentObject is implicit
    newNode->previous = this; // a must
    next->previous = newNode;
    next = newNode;
}
```



Static Data Members

- Using global variables in a class is not a good OOP practice.
  - No encapsulation
  - No information hiding/access control
- A static data member has *only* one copy shared by all objects in the class.

```
// int gLastID = 0; // You can use global but not OOP
class AntsT {
  public:
    AntsT();
    int getID() const;
  private:
    static int lastID; // this variable has only one copy
    int id;
};

int Ants::lastID = 0; // scope resolution operator
AntsT::AntsT() :id(lastID) {
    lastID++; // scope resolution operator not needed
}
```

Static Member Functions

- Functions can also be declared *static*, but static functions can only access static data members.
- If a static function is public, it can be accessed without referencing to a particular object.

```
class AntsT {
  public:
   AntsT();
   int getID() const;
  private:
   int id;
   static int lastID;
   static int getNewID();
   static int incrementID();
};
int AntsT::getNewID() {
   return lastID;
}
int AntsT::incrementID() {
   return lastID++;
}
```

```
class Math {
  public:
   ...
   static double sin(double);
   static double cos(double);
   ...
};

void main() {

   double x = Math::sin(1.0);
   double y = Math::cos(2.0);
}
```

Convert Constructors

- A constructor that can be used for typecasting.

```
class TimeT {
  public:
    TimeT();
    TimeT(int rawMinutes);
    TimeT(int minutes, int hours);
  private:
    int minutes;
    int hours;
};
TimeT::TimeT(int rawMinutes) {
    hours = rawMinutes/60;
    minutes = rawMinutes%60;
}
int main() {
    int x = 123;
    TimeT time1(10, 10), time2(123), time3, time4;
    time3 = TimeT(x);  // use as a type cast function
    time4 = x;         // implicit cast works as well
} // explicit cast is a better style
```

Copying Objects

- You can assign one object to another of the same type.

```
class GradesT {
  public:
    GradesT();
    GradesT(int score);
    int getScore();
  private:
    int score;
}
int main() {
    GradesT student1(95), student2;
    GradesT student3 = student1;
    student2 = student1;
}
```

Result: score's in all three objects are 95 now.

- Default copying works by *memberwise assignment* like `struct`.
- When are objects copied?
  - Passed-by-value to another function
  - Returned by a function
  - Assignment

Problems With Shallow Copying

- Shallow copying: member-wise copying
- Deep copying

```
class StringT {
  public:
    StringT() {str=NULL;};
    StringT(const char *inputData);
    ~StringT() {delete [] str;}
    void setString(const char *inputData);
    char *getString() const {return str;}
  private:
    char *str;
};

StringT::StringT(const char *inputData) {
    str = new char[strlen(inputData)+1];
    strcpy(str, inputData);
}
void StringT::setString(const char *inputData) {
    if (strlen(inputData)<=strlen(str))
        strcpy(str, inputData);
    else
        str = strdup(inputData);
}
```

Problems With Shallow Copying Continued (code)

- What kinds of problems do we have in the following code?

```
int main() {
    StringT string1("Hello");
    StringT string2 = string1;
    StringT *string3 = new StringT();

    *string3 = string1;
    cout << string1.getString() << "\n";
    cout << string2.getString() << "\n";
    string2.setString("OK");
    cout << string1.getString() << "\n";
    delete string3;
    cout << string2.getString() << "\n";
}
```

Output:

```
Hello
Hello
OK
segmentation fault.
```

- Reason: all objects share the same `str` pointer due to memberwise (shallow) copying.

Copy Constructor

- A copy constructor is a function that is automatically invoked when an object is being copied.

```
class StringT {
  public:
    StringT(const StringT &originalObject);
    ...
};
StringT::StringT(const StringT &originalObject) {
    int length = strlen(originalObject.getString());
    str = new char[length+1];
    strcpy(str, originalObject.getString());
}
int main() {
    StringT string1("Hello"), string2;
    StringT string3 = string1; // call copy constructor
    string2 = string1; // still do memberwise copying
}
```

Why? Constructor are called only when a variable is being constructed.

Solution: Overload the assignment operator (later on this topic)

Rule of three

A rule of thumb in C++ (prior to C++11) that claims that if a class defines any of the following then it should probably explicitly define all three

- **Destructor** – call the destructors of all the object's class-type members
- **Copy constructor** – construct all the object's members from the corresponding members of the copy constructor's argument, calling the copy constructors of the object's class-type members, and doing a plain assignment of all non-class type (e.g., *int* or pointer) data members
- **Copy assignment operator** – assign all the object's members from the corresponding members of the assignment operator's argument, calling the copy assignment operators of the object's class-type members, and doing a plain assignment of all non-class type (e.g. *int* or pointer) data members.