C++
Programming Bootcamp 2

COSC2802

Topic 11 **Streams (part 2)**

Technical Details

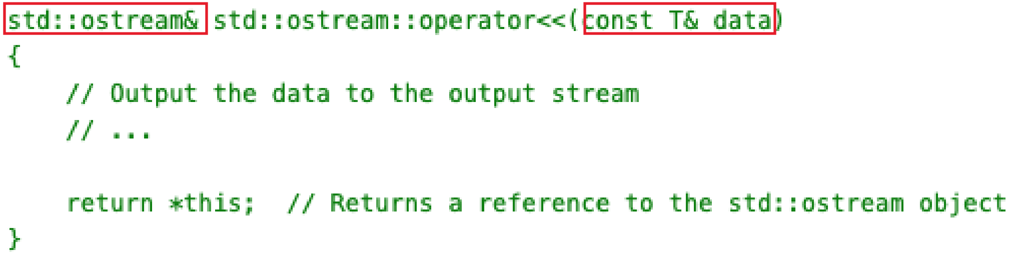
Remainder of the slides introduces material which gives further detail to supplement content in zybooks

Technical Details

- The >> read operator only reads the ***next valid token*** from the input stream
  - The validity of a token is based on the type of the variable that is being read
  - Token are *always* separated by whitespace
  - Whitespace is not read!
    - This includes spaces and newlines
    - Even if `std::string` is used, whitespace is ignored

How does << (and >>) work?

Output operator defined as (called on object of a stream class):

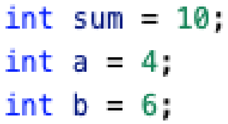


```
std::ostream& std::ostream::operator<<(const T& data)
{
    // Output the data to the output stream
    // ...

    return *this;  // Returns a reference to the std::ostream object
}
```

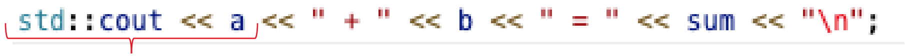
- T is some type
- Note: returns a reference to the output stream

Chaining of operators << and >>



```
int sum = 10;
int a = 4;
int b = 6;
```

Consider the following chaining of insertion operators:



```
std::cout << a << " + " << b << " = " << sum << "\n";
```

How does it work?

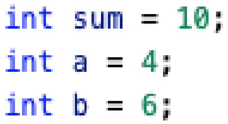
roughly equivalent to:
std::cout.operator<<(a)

- Leftmost operation evaluated first – left hand *std::cout*, right hand an int *a*
  - It prints the value of the right hand operand, *a* (defined for built-in types such as int)
  - Returns a reference to the left hand operand, *std::cout*

- Next *std::cout* << " + " is evaluated: prints the string (defined for strings), and again returns reference to left hand operand *std::cout*.

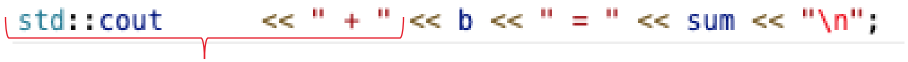
- Process continues:
  - Successively evaluating next expression
  - Each time returning reference to the left hand operand - in this case *std::cout*

Chaining of operators << and >>



```
int sum = 10;
int a = 4;
int b = 6;
```

Consider the following chaining of insertion operators:



```
std::cout        << " + " << b << " = " << sum << "\n";
```

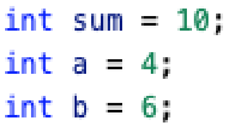
How does it work?

- Leftmost operation evaluated first – left hand *std::cout*, right hand an int *a*
  - It prints the value of the right hand operand, *a* (defined for built-in types such as int)
  - Returns a reference to the left hand operand, *std::cout*

- Next *std::cout* << " + " is evaluated: prints the string (defined for strings), and again returns reference to left hand operand *std::cout*.

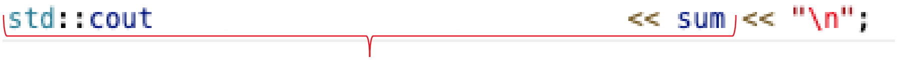
- Process continues:
  - Successively evaluating next expression
  - Each time returning reference to the left hand operand - in this case *std::cout*

Chaining of operators << and >>



```
int sum = 10;
int a = 4;
int b = 6;
```

Consider the following chaining of insertion operators:



```
std::cout                        << sum << "\n";
```

How does it work?

- Leftmost operation evaluated first – left hand *std::cout*, right hand an int *a*
  - It prints the value of the right hand operand, *a* (defined for built-in types such as int)
  - Returns a reference to the left hand operand, *std::cout*

- Next *std::cout* << " + " is evaluated: prints the string (defined for strings), and again returns reference to left hand operand *std::cout*.

- Process continues:
  - Successively evaluating next expression
  - Each time returning reference to the left hand operand - in this case *std::cout*

What happens if: stream fully consumed; or invalid input (token can't be converted/assigned)?

Overloading Stream Operators

Stream Operators | user-defined classes

Stream Insertion and Extraction operators (<< and >>):

- Defined for built-in types and some standard library types
- For user-defined classes need to explicitly overload them

Why overload stream operators?

- Easier input and output
  - Use cin and cout for output
  - Chaining of << and >> operators
- Control over how objects are formatted when sent to stream
- Readability – code is more consistent/readable to others (flow of data is more obvious).
- …

Overloading Stream Operators (zybooks)

Distinction between:

- **Class member** function overloading:
  - allows operator to be called **directly on the object** itself (single parameter)
- **non-member function** overloading:
  - when working with classes not under your control/can’t be changed (e.g. STL)

Recommended:

- overload the output (<<), input operator (>>) as **non-member function**
  - don't have direct control over the *std::cout* object itself (part of the STL)

1. Member function Overloading << and >>

- Allows the operator to be directly called on the object itself
- Member functions automatically bound to the operand on the left-hand side
- Useful when the operation is tied to the class
- …

1. Member function Overloading << and >>

Example: Extending the use of << and >> to new functions.

We have a waiting list class: it takes in peoples names and keeps a track of their order.

```
class WaitingLine {
  public:
    std::queue<std::string> line;
    WaitingLine& operator<<(const std::string& name) {
            this->line.push(name);
            return *this;
    }
    WaitingLine& operator>>(std::string& frontname) {
            frontname = line.front();
            this->line.pop();
            return *this;
    }
};
```

```
int main() {
    std::string name;
    WaitingLine line;
    for(int i = 0; i < NUM_PEOPLE; ++i) {
        std::cin >> name;
        line << name; // unconventional use of <<
    }
    for(int i = 0; i < NUM_PEOPLE; ++i) {
        line >> name; // unconventional use of >>
        std::cout << "[" << i << "] : " << name << std::endl;
    }
    return EXIT_SUCCESS;
}
```

2. Non-member function overloading

Example: Extending cin and cout to work with custom classes

Why?

- User defined class doesn’t work with cin and cout
- Allows chaining of output to cout (and input from cin)
- Standard usage: the left-hand operand for << and >> is usually a stream
- …

Aside: Friend Functions and Classes

- allows non-member functions to access **private** or **protected** members of a clas

2. Non-member function overloading

Example: Extending cin and cout to work with custom classes

```
class Coordinate {
    private:
        int x, y;
    public:
        Coordinate(int x = 0, int y = 0) { this->x = x; this->y = y; }
        friend std::ostream & operator<< (std::ostream &out, const Coordinate &c);
        friend std::istream & operator>> (std::istream &in,  Coordinate &c);
};

std::ostream & operator<< (std::ostream &out, const Coordinate &c) {
    out << c.x << "," << c.y << std::endl;
    return out;
}

std::istream & operator>> (std::istream &in,  Coordinate &c) {
    std::cout << "Enter X ";
    in >> c.x;
    std::cout << "Enter Y ";
    in >> c.y;
    return in;
}
```

```
int main() {
   Coordinate c1;
   std::cin >> c1;
   std::cout << "The coordinate is located at: " << c1;
   return 0;
}
```

2. Non-member function overloading

Example: Extending cin and cout to work with custom classes

```
class Coordinate {
    private:
        int x, y;
    public:
        Coordinate(int x = 0, int y = 0) { this->x = x; this->y = y; }
        friend std::ostream & operator<< (std::ostream &out, const Coordinate &c);
        friend std::istream & operator>> (std::istream &in,  Coordinate &c);
};

std::ostream & operator<< (std::ostream &out, const Coordinate &c) {
    out << c.x << "," << c.y << std::endl;
    return out;
}

std::istream & operator>> (std::istream &in,  Coordinate &c) {
    std::cout << "Enter X ";
    in >> c.x;
    std::cout << "Enter Y ";
    in >> c.y;
    return in;
}
```

```
int main() {
   Coordinate c1;
   std::cin >> c1;
   std::cout << "The coordinate is located at: " << c1;
   return 0;
}
```

Friend Functions and Classes

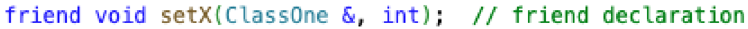
Friend function of a class is a **non-member function of that class** that:

- Can access both **public** and **non-public** class members
- Can be functions, entire classes or member functions of other classes

Declaring a Friend

To declare **a non-member function** as a friend of a class:

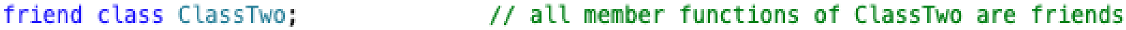
- Put function prototype in **class definition,** preceded by keyword **friend**



```
friend void setX(ClassOne &, int);  // friend declaration
```

To declare **all member functions of a class** as friends of another declare:

- all member functions of *ClassTwo* as friends of *ClassOne* put declaration in the definition of class *ClassOne* e.g:



```
friend class ClassTwo;                // all member functions of ClassTwo are friends
```

NOTE:

- Friendship must be explicitly declared:
  - for class B to be friend of class A, class A must explicitly declare class B as friend
- Friendship is not symmetric:
  - if class A is a friend of class B, cannot infer that class B is a friend of class A.
- Friendship is not transitive:
  - if class A is friend of class B, class B is friend of class C, **cannot infer** class A is a friend of class C

Controlled use of friend declarations

Declaring the whole class as a friend: for closely related classes that need comprehensive access to internal details. Example **game simulation** where:

- GameEngine class manages the state of a game,
- Debugger class, as a friend, helps monitor internal state during development/testing

Non-member functions as friends: for specific operations like comparisons or manipulations across multiple objects. Example:

- a compare function that compares the private members of two objects (e.g. area)

Example: whole class as friend

```
#include <iostream>
#include <string>
// Forward declaration of Debugger
class Debugger;
class GameEngine {
private:
  int score = 0;
  std::string level = "Level 1";
  std::string gameState = "Running";
  friend class Debugger; // Class as friend
public:
  void startGame() { gameState = "Started"; }
  void advanceLevel() {
    level = "Level 2";
    score += 100;
  }
  void endGame() { gameState = "Game Over"; }
};
```

```
class Debugger {
public:
  void logGameState(const GameEngine& engine) {
    // Access private data of GameEngine
    std::cout << "[DEBUG] : " <<
       engine.gameState << "\n";
    std::cout << "[DEBUG] Current Level: " <<
      engine.level << "\n";
    std::cout << "[DEBUG] Current Score: " <<
      engine.score << "\n";
  }

  void simulateCrash(const GameEngine& engine) {
  // simulate a game crash
  // needs to access private data of GameEngine
  }
};
```

Why?

- Might simplify design when multiple functions need broad access to members

But

- Avoid overuse: could lead to unintentional misuse; hard to debug; …

Example: non-member friend function

```
 #include <iostream>
class Rectangle {
  private: int width, height;
  // Declare compare as a friend function
  friend bool compareArea(const Rectangle& rect1, const Rectangle& rect2);

public:
  Rectangle(int w, int h) : width(w), height(h) {}
  void display() const {
    std::cout << "Rectangle (width: " << width << ", height: " << height << ")\n";
  }
};

// Non-member friend function
  bool compareArea(const Rectangle& rect1, const Rectangle& rect2) {
  // Access private members of both Rectangle objects
  int area1 = rect1.width * rect1.height;
  int area2 = rect2.width * rect2.height;
  return area1 > area2; // Return true if rect1 is larger
}
```

Why?

- Logically external to core behaviour of rectangle
- Symmetry in comparison: `compareArea(rect1, rect2);` vs rect1.compareArea(rect2);

Comparison: Overloading member vs non-member

Feature	Member Function	Non-Member Function
Definition	Defined within the class	Defined outside the class
Tied to Object	Yes, operates on an instance of the class	No, operates independently of any specific object
Access to this	Has access to this pointer	No access to this pointer
Access Scope	Can access all members (private, protected, public)	Can only access public members via methods (*friend)
Number of Arguments	Takes one fewer argument i.e. implicit this pointer	Requires all operands explicitly e.g., lhs, rhs
Calling Method	Called via an object e.g., obj.func()	Called independently e.g., func(lhs, rhs)
Use Case	Used for instance-specific operations	Used for operations not tied to a particular object
Comparison Operators	Can be overloaded to compare the object itself	Often better suited for symmetric comparisons