# Object-Oriented Programming: Operator Overloading

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

# Operator Overloading

- Operator overloading
- Complication with overloading '+'
- Extending the set of overloaded operators
- Overloading []
- Overloading assignment '='
- Overloading ()
- Overloading `new` and `delete`
- Type conversion operator
- Odds and ends

# Operator Overloading

- You can alter the behavior of operators.

```
class MenuItemT {
  public:
    MenuItemT(double itemPrice, char *itemName);
    MenuItemT(const MenuItemT &object);
    ~MenuItemT();
    double getPrice() const;
    char *getName() const;
    double operator+(MenuItemT secondItem) const;//overloading
  private:
    double price;
    char *fName;
};
double MenuItemT::operator+(MenuItemT secondItem) const {
    return getPrice() + secondItem.getPrice();
}
void main() {
    MenuItemT item1(65,"Big Mac"), item2(35,"Fresh Fries");
    cout << "The total is NT$" << item1+item2 << ".\n";
}
```

**keyword `operator` is required**

**Output: The total is NT$100.**

# Complications With Overloaded Operators

```
void main() {
    MenuItemT item1(65,"Big Mac");
    MenuItemT item2(35,"Fresh Fries");
    MenuItemT item3(30,"Coke");
    double total = item1 + item2 + item3;
}
```

**Error:**
**illegal operand**

**Explanation:**
`item1 + item2` returns a double,
so you have **`double + item3`**

```
// proposed solution #1
double MenuItemT::operator+(double currentTotal) {
    return currentTotal + getPrice();
}
```

**Won't work**

```
// proposed solution #2
double MenuItemT::operator+(double currentTotal, MenuItemT item2){
    return currentTotal + item2.getPrice();
}
```

**Won't work either**

# Complications With Operator Overloading Continued

- A proposed solution

```
// proposed solution #3: top-level function
double operator+(double currentTotal, MenuItemT item2){
    return currentTotal + item2.getPrice();
}
```

- But the following code will still fail.

```
item1 + (item2 + item3);  // Error: illegal operand
// Explanation: item + double
```

- A correct solution for handling all three situations:

```
// overload #2
double operator+(double currentTotal, MenuItemT item2) {
    return currentTotal + item2.getPrice();
}
// overload #3
double MenuItem::operator+(double currentTotal) {
    return currentTotal + getPrice();
}
```

# Complications With Operator Overloading: Alternative

- Write the class with a *conversion constructor* and *no overloaded operator* member functions.

```
class MenuItemT {
  public:
    MenuItemT(double itemPrice, char *itemName);
    MenuItemT(double x);
    MenuItemT(const MenuItemT &object);
    ~MenuItemT();
    double getPrice() const;
    char *getName() const;
  private:
    double price;
    char *name;
}
MenuItemT::MenuItemT(double x) {
    price = x;
    name = NULL;
}
double operator+(MenuItemT item1, MenuItemT item2) {
    return item1.getPrice() + item2.getPrice();
}
```

# Improper Use of Operator Overloading

- Overloading of mathematical operators make more sense for mathematical objects. Do not abuse them.

```
// improper overloading
s + 5;     // stands for s.push(5)
x = s--;   // stands for x = s.pop()
```

- Overloading obscure operators can be dangerous.

```
// assume that you redefine ^ to mean "power"
int x=5;
int y = x^2 + 1; // expecting 26 but you get 125, why?
```

- You cannot overload existing built-in operator functions.

```
int operator+(int number1, int number2) {
    return number1 - number2;
}
```

**Error: cannot overload built-in operator function**

# Providing a Full Set of Related Operators

- If you provide a + operator, you should also provide related operators such as += and ++

```
class TimeT {
  public:
    TimeT();
    TimeT(int hours, int minutes, int seconds);
    void display();
    TimeT operator+(const TimeT &secondTime);
    TimeT &operator+=(TimeT secondTime);// addition+assignment
    TimeT &operator+=(int num);          // addition+assignment
    TimeT operator*(int num);          // multiply
    TimeT &operator*=(int num);        // multiply+assignment
    TimeT operator++(int);             // postfix format
    TimeT &operator++();               // prefix format
  private:
    int hours;
    int minutes;
    int seconds;
    void normalizeTime();
}
```

# Providing a Full Set of Related Operators

```
TimeT TimeT::operator+(const TimeT &time2) {
    TimeT tempTime(hours+time2.hours, minutes+time2.minutes,
                   seconds+time2.seconds);
    return tempTime;
}
TimeT &TimeT::operator*=(int num){
    hours *= num;
    minutes *= num;
    seconds *= num;
    normalizeTime();
    return *this;
}
TimeT TimeT::operator++(int) {
    TimeT tempTime = *this;
    seconds++;
    normalizeTime();
    return tempTime;
}
TimeT &TimeT::operator++() {
    seconds++;
    normalizeTime();
    return *this;
}
```

```
int main() {
    TimeT firstTime(1,1,3);
    TimeT secondTime;
    secondTime = firstTime++;
    firstTime.display();
    secondTime.display();

    secondTime = ++firstTime;
    firstTime.display();
    secondTime.display();
}
```

**Output:**

**1:1:4**
**1:1:3**
**1:1:5**
**1:1:5**

# Overloading []

```
class ArrayT {
  public:
    ArrayT();
    ArrayT(int arraySize);
    ~ArrayT();
    // void insertElement(int element, int slot);
    // int getElement(int slot) const;
    int &operator[](int slot);
  private:
    int arraySize;
    int *elements;
};
int &ArrayT::operator[](int slot) {
    if (slot>=0 && slot <arraySize)
        return elements[slot];
    cerr << "Subscript out of range\n";
    return elements[0];
}
int main() {
    ArrayT array(20);
    array[1] = 10;
    cout << array[1] << " " << array[20];
}
```

**Output:**

**10 Subscript out of range 0**

# Overloading Assignment [code]

- Recall that in the `StringT` class, the copy constructor is not called in the following situation.

```
StringT string1("Hello");
StringT string2;
string2 = string1;
```

- You can overload the assignment operator to do deep copying.

```
StringT &StringT::operator=(const StringT &orignalObject) {
    int length;
    if (this == &orignalObject) // why do we need this?
        return *this;

    delete [] str;
    length = strlen(orignalObject.str);
    str = new char[length+1];
    strcpy(str, orignalObject.str);
    return *this; // why do we need to return?
}
```

# 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.

# Operators Related to Comparison

- If you overload assignment, your clients might expect you to overload equality, inequality, and relative operators.

```
bool StringT::operator==(const StringT &secondObject) const {
    return strcmp(str, secondObject.str) == 0;
}
bool StringT::operator!=(const StringT &secondObject) const {
    return strcmp(str, secondObject.str) != 0;
}
bool StringT::operator<(const StringT &secondObject) const {
    return strcmp(str, secondObject.str) < 0 ;
}
bool StringT::operator>(const StringT &secondObject) const {
    return strcmp(str, secondObject.str) > 0 ;
}
```

# Overloading ()

- Overloading () can make objects behave like functions.

```
class PolynomialT {
  public:
    PolynomialT(double order2, double order1, double order0);
    double operator() (double x);
  private:
    double coef[3];
};
PolynomialT::PolynomialT(double order2, double order1,
                         double order0) {
    coef[2]=order2;
    coef[1]=order1;
    coef[0]=order0;
}
double PolynomialT::operator()(double x) {
    return coef[2]*x*x + coef[1]*x + coef[0];
}
Int main() {
    PolynomialT f(2, 3, 4);
    cout << f(2);
}
```

**Output: 18**

# Overloading () with Multiple Arguments

- Suppose that you have a matrix class as follows and you would like to access the elements without using accessor functions.

```
class MatrixT {
  public:
    MatrixT(int dim1, int dim2);
    ~MatrixT();
  private:
    int **matrix;
    int dim1;
    int dim2;
};
```

```
// OK
int *operator[](int x);
// Illegal
int &operator[][](int x);
```

- Another way is to overload () operator.

```
int &MatrixT::operator()(int x, int y) {
    if (x>=0 && x<dim1 && y>=0 && y<dim2)
        return matrix[x][y];
    cout << "out of bounds\n";
    return matrix[0][0];
}
```

# ~~Smart Pointers~~ Class member access

- To access a member function of a sub-object to the main object.

```
class NameT {
  public:
    NameT(const char *name);
    ~NameT();
    char *getName();
  private:
    char *name;
};
class PersonT {
  public:
    PersonT(const char *name, int age);
    int getAge();
    NameT *operator->();
  private:
    NameT *nameObject; // must be pointer
    int age;
};
NameT *PersonT::operator->() {
    return nameObject;
}
```

```
int main() {
    PersonT person("John",12);
    cout << person->getName();
    // PersonT *personPtr;
    // personPtr = &person;
    // personPtr->getName();
}
```

**Output: john**

**//how**

An expression **x->m** is interpreted as

**(x.operator->())->m**

**Error:**
**'getname' is not a struct/union/class member**

# Overloading `new` and `delete`

- You can take control of memory management locally for a class or at the top-level for all variables

```
class RandomT {
  public:
    Random(int data);
    int getData();
    void *operator new(size_t objectSize);
    void operator delete(void *object);
  private:
    int data;
};
void *RandomT::operator new(size_t objectSize) {
    cout << "new\n";
    return malloc(objectSize);
}
void RandomT::operator delete(void *object) {
    cout << "delete\n";
    free(object);
}
```

**Note:**
**parameter and return types are required.**

# Type Conversion Operators

- You can convert from ANSI C strings to the `StringT` class through the convert constructor.
- How about the other way from `StringT` to the ANSI C strings?

```
class StringT {
  public:
    StringT();
    StringT(char *inputData); // convert constructor
    StringT(const StringT &originalObject);
    ~String();
    char *getString() const;
    operator const char*() const;
  private:
    char *str;
}
StringT::operator const char *() const {
    return str;
}
void main() {
    StringT string1 = StringT("Hello");
    cout << strlen(string1) << " " <<string1 <<"\n";
}
```

**Note:**
**no return type but it does return something.**

**Output:**
**5 Hello**

# Overloading <<

```
class Point {
public:
    int x, y;

    Point() {
        x = y = 0;
    }

    Point(int x, int y) {
        this->x = x;
        this->y = y;
    }
};

ostream &operator<<(ostream &s, Point p) {
    s << "("<< p.x << ", " << p.y << ")";
    return s;
}
```

# operator overloading:

1. operator op

   op: any of the following operators:

   - + - * / % ^ & | ~ ! = < >
   - += -= *= /= %= ^= &= |=
   - << >> >>= <<= == != <= >=  (<=> since C++20)
   - && || ++ -- , ->* -> ( ) [ ]

2. operator type
3. operator new / operator new []
4. operator delete / operator delete []

# Odds and Ends on Operator Overloading

- Overloading unary operators

```
PointT PointT::operator-() const {
    PointT temp(-x,-y);
    return temp;
}
```

- Can you overload every operator?

  No, not there listed here.

```
.   *   :: ?: sizeof()
```

- Can you create new operators?

  No, only built-in operators.

- Conclusion:

  Do not abuse operator overloading. You should be able to live without it.