Programming Design

Operator Overloading

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Outline

- **Motivations and prerequisites**
- Overloading comparison and indexing operators
- Overloading assignment and self-assignment operators
- Overloading addition operators

Recall our MyVector class

```
class MyVector
{
private:
  int n;
  double* m;
public:
  MyVector();
  MyVector(int n, double m[]);
  MyVector(const MyVector& v);
  ~MyVector();
  void print();
};
```

Recall our MyVector class

```
MyVector::MyVector()
{
  n = 0;
  m = nullptr;
}
MyVector::MyVector(int dim, double v[])
{
  n = dim;
  m = new double[dim];
  for(int i = 0; i < dim; i++)
    m[i] = v[i];
}
MyVector::~MyVector()
{
  delete [] m;
}
```

```
MyVector::MyVector(const MyVector& v)
{
  n = v.n;
  m = new double[n];
  for(int i = 0; i < n; i++)
    m[i] = v.m[i];
}
void MyVector::print()
{
  cout << "(";
  for(int i = 0; i < n - 1; i++)
    cout << m[i] << ", ";
  cout << m[n-1] << ")\n";
}
```

Comparing `MyVector` objects

- When we have many vectors, we may need to **compare** them.
- For vectors u and v:
  - $u = v$ if their dimensions are equal and $u_i = v_i$ for all i.
  - $u < v$ if their dimensions are equal and $u_i < v_i$ for all i.
  - $u \leq v$ if their dimensions are equal and $u_i \leq v_i$ for all i.
- How to add **member functions** that do comparisons?
  - Naturally, they should be **instance** rather than static functions.

Member function `isEqual()`

```
class MyVector
{
private:
  int n;
  double* m;
public:
  MyVector();
  MyVector(int n, double m[]);
  MyVector(const MyVector& v);
  ~MyVector();
  void print();
  bool isEqual(const MyVector& v);
};
```

```
bool MyVector::isEqual(const MyVector& v)
{
  if(n != v.n)
    return false;
  else
  {
    for(int i = 0; i < n; i++)
    {
      if(m[i] != v.m[i])
        return false;
    }
  }
  return true;
}
```

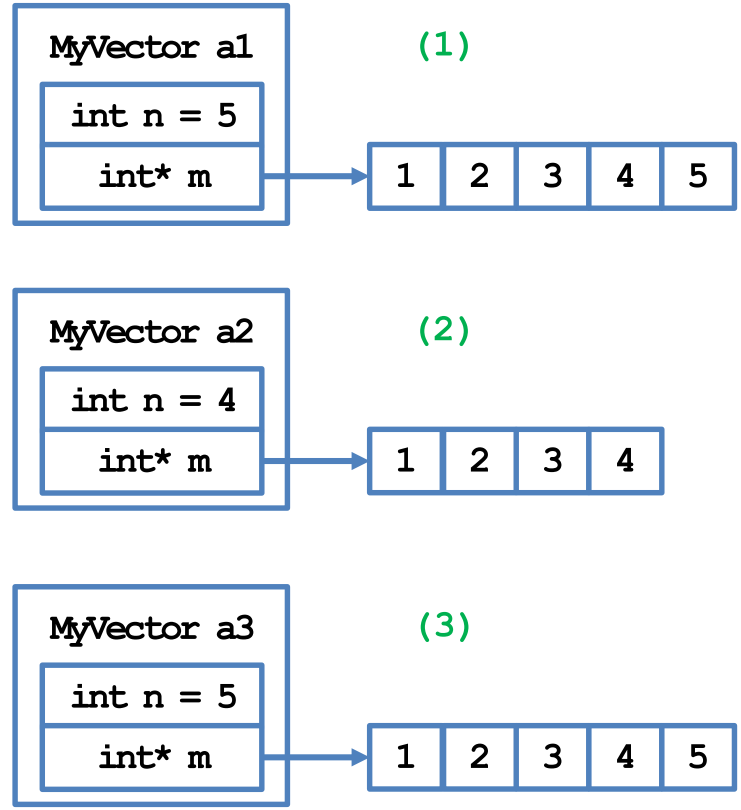
Member function `isEqual()`

```
int main()
{
  double d1[5] = {1, 2, 3, 4, 5};
  MyVector a1(5, d1); // (1)

  double d2[4] = {1, 2, 3, 4};
  MyVector a2(4, d2); // (2)
  MyVector a3(a1); // (3)

  cout << a1.isEqual(a2) ? "Y\n" : "N\n"; // N
  cout << a1.isEqual(a3) ? "Y\n" : "N\n"; // Y

  return 0;
}
```



`isEqual()` is fine, but ...

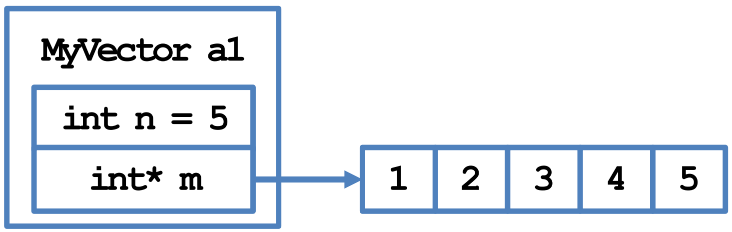
- Adding the instance function **`isEqual()`** is fine.
  - But it is not the most intuitive way.
  - If we can write **`if(a1 == a2)`**, it will be great!
- Of course we cannot:
  - The compiler does not know what to do to this statement.
  - We need to define **==** for **`MyVector`** just as we define member functions.
- In fact, **==** has been **overloaded** for different data types.
  - We may compare two **`int`**s, two **`double`**s, one **`int`** and one **`double`**, etc.
  - We will now define how **==** should compare two **`MyVector`**s.
- This is **operator overloading**.

Operator overloading

- Most operators (if not all) have been overloaded in the C++ standard.
  - E.g., the division operator `/` has been overloaded.
  - Divisions between integers differ from those between fractional values!
- Overloading operators for self-defined classes are **not required**.
  - Each overloaded operator can be replaced by an instance function.
  - However, it may make programs **clearer** and the class **easier to use**.
- Some **restrictions**:
  - Not all operators can be overloaded (see your textbook).
  - The number of operands for an operator cannot be modified.
  - One cannot create new operators.

`this`

- When you create an object, it occupies a memory space.



- Inside an instance function, **`this`** is a **pointer** storing the **address** of that object.
  - **`this`** is a C++ keyword.

```
class A
{
private:
  int a;
public:
  void f() { cout << this << "\n"; }
  A* g() { return this; }
};
int main()
{
  A obj;
  cout << &obj << "\n"; // 0x9ffe40
  obj.f(); // 0x9ffe40
  cout << (&obj == obj.g()) << "\n"; // 1
  return 0;
}
```

this

- The two implementations are identical:

```
void MyVector::print()
{
  cout << "(";
  for(int i = 0; i < n - 1; i++)
    cout << m[i] << ", ";
  cout << m[n - 1] << ")\n";
}
```

```
void MyVector::print()
{
  cout << "(";
  for(int i = 0; i < this->n - 1; i++)
    cout << this->m[i] << ", ";
  cout << this->m[this->n - 1] <<
")\n";
}
```

Why using this?

- Suppose that **n** is an instance variable.
  - Usually you can use **n** directly instead of **this->n**.
  - However, if you want to have a **local variable** or **function parameter** having the same name as an instance variable, you need **this->**.

```
MyVector::MyVector(int d, int v[])
{
  n = d;
  for(int i = 0; i < n; i++)
    m[i] = v[i];
}
```

```
MyVector::MyVector(int n, int m[])
{
  this->n = n;
  for(int i = 0; i < n; i++)
    this->m[i] = m[i];
}
```

- A local variable hides the instance variable with the same name.
  - **this->n** is the instance variable and **n** is the local variable.

Good programming style

- You may choose to always use **`this->`** when accessing instance variables and functions.
- This will allow other programmers (or yourself in the future) to know they are members without looking at the class definition.
- Some other reasons for using **`this`** will become clear shortly.

Constant objects

- Some variables are by nature **constants**.

```
const double PI = 3.1416;
```

- We may also have **constant objects**.

```
const MyVector ORIGIN_3D(3, 0);
```

  – This is the origin in $\mathbf{R}^3$. It should not be modified.
- Should there be any restriction on **instance function invocation**?

Constant objects

- A constant object cannot invoke a function that modifies its instance variables.
  - In C++, functions that may be invoked by a constant object must be declared as a **constant instance function**.
- For a constant instance function:
  - It can be called by non-constant objects.
  - It cannot modify any instance variable.
- For a non-constant instance function:
  - It cannot be called by constant objects even if no instance variable is modified.

```
class MyVector
{
private:
  int n;
  int* m;
public:
  MyVector();
  MyVector(int dim, int v[]);
  MyVector(const MyVector& v);
  ~MyVector();
  void print() const;
};
```

Constant instance variables

- We may have **constant instance variables**.
  - E.g., for a vector, its dimension should be fixed once it is determined.
- Obviously, a constant instance variable should be initialized in the constructor(s).
  - However:

```
MyVector::MyVector()
{
  n = 0; // error!
  m = nullptr;
}
```

- A constant instance variable cannot be assigned a value (locally or globally).

```
class MyVector
{
private:
  const int n;
  int* m;
public:
  MyVector();
  MyVector(int dim, int v[]);
  MyVector(const MyVector& v);
  ~MyVector();
  int getN() const;
  int getM() const;
  void print() const;
};
```

Member initializers

- We need a **member initializer**.
  - A specific operation for initializing an instance variable.
  - Can also be used for initializing non-constant instance variables.
- Member initializers are used a lot in general.

```
MyVector::MyVector() : n(0)
{
  m = nullptr;
}
MyVector:: MyVector(int dim, int v[]) : n(dim)
{
  for(int i = 0; i < n; i++)
    m[i] = v[i];
}
MyVector:: MyVector(const MyVector& v) : n(v.n)
{
  m = new double[n];
  for(int i = 0; i < n; i++)
    m[i] = v.m[i];
}
```

Outline

- Motivations and prerequisites
- **Overloading comparison and indexing operators**
- Overloading assignment and self-assignment operators
- Overloading addition operators

Overloading an operator

- An operator is overloaded by "implementing a **special instance function**".
  - It cannot be implemented as a static function.
- Let ***op*** be the operator to be overloaded, the "special instance function" is always named

```
operatorop
```

  - The keyword `operator` is used for overloading operators.
- Let's overload **==** for **`MyVector`**.

Overloading ==

- Recall that we defined **`isEqual()`**:

```
class MyVector
{
private:
  int n;
  double* m;
public:
  // others
  bool isEqual
    (const MyVector& v) const;
};
```

```
bool MyVector::isEqual
  (const MyVector& v) const
{
  if(this->n != v.n)
    return false;
  else {
    for(int i = 0; i < n; i++) {
      if(this->m[i] != v.m[i])
        return false;
    }
  }
  return true;
}
```

Overloading ==

- To overload ==, simply do this:

```
class MyVector
{
private:
  int n;
  double* m;
public:
  // others
  bool operator==
    (const MyVector& v) const;
};
```

```
bool MyVector::operator==
  (const MyVector& v) const
{
  if(this->n != v.n)
    return false;
  else {
    for(int i = 0; i < n; i++) {
      if(this->m[i] != v.m[i])
        return false;
    }
  }
  return true;
}
```

Invoking overloaded operators

- We are indeed implementing instance functions with special names.
- Regarding **invoking** these instance functions:

```
int main() // without overloading
{
  double d1[5] = {1, 2, 3, 4, 5};
  const MyVector a1(5, d1);

  double d2[4] = {1, 2, 3, 4};
  const MyVector a2(4, d2);
  const MyVector a3(a1);

  cout << a1.isEqual(a2) ? "Y\n" : "N\n";
  cout << a1.isEqual(a3) ? "Y\n" : "N\n";

  return 0;
}
```

```
int main() // with overloading
{
  double d1[5] = {1, 2, 3, 4, 5};
  const MyVector a1(5, d1);

  double d2[4] = {1, 2, 3, 4};
  const MyVector a2(4, d2);
  const MyVector a3(a1);

  cout << a1 == a2 ? "Y\n" : "N\n";
  cout << a1 == a3 ? "Y\n" : "N\n";

  return 0;
}
```

Invoking overloaded operators

- Interestingly, we may also do:

```
int main() // with overloading
{
  double d1[5] = {1, 2, 3, 4, 5};
  const MyVector a1(5, d1);

  double d2[4] = {1, 2, 3, 4};
  const MyVector a2(4, d2);
  const MyVector a3(a1);

  cout << a1.operator==(a2) ? "Y\n" : "N\n";
  cout << a1.operator==(a3) ? "Y\n" : "N\n";

  return 0;
}
```

Overloading <

- Let’s overload **<**:

```
class MyVector
{
private:
  int n;
  double* m;
public:
  bool operator==
    (const MyVector& v) const;
  bool operator<
    (const MyVector& v) const;
};
```

```
bool MyVector::operator<
  (const MyVector& v) const
{
  if(this->n != v.n)
    return false;
  else {
    for(int i = 0; i < n; i++) {
      if(this->m[i] >= v.m[i])
        return false;
    }
  }
  return true;
}
```

Overloading !=

- To overload **!=**, let's utilize the overloaded **==**:

```
class MyVector
{
  // ...
  bool operator==
    (const MyVector& v) const;
  bool operator!=
    (const MyVector& v) const;
};
```

```
bool MyVector::operator!=
  (const MyVector& v) const
{
  if(*this == v)
    return false;
  else
    return true;
  // or return !(*this == v);
}
```

- How would you overload **>=**?

Parameters for overloaded operators

- The **number of parameters** is **restricted** for overloaded operators.
  - The **types of parameters** are not restricted.
  - The **return type** is not restricted.
- **What is done** is not restricted.
  - Always avoid unintuitive implementations!

```
class MyVector
{
  // ...
  bool operator==(const MyVector& v) const;
  bool operator==(MyVector v) const;
  bool operator==(int i, int j); // error
};
```

```
class MyVector
{
  // ...
  void operator==(int i) const
  {
    cout << "...\n";
  } // no error but never do this!
};
```

Overloading the indexing operator

- Another natural operation that is common for vectors is indexing.
  - Given vector v, we want to know/modify the element v_i.
- For C++ arrays, we use the indexing operator `[]`.
- May we overload `[]` for **`MyVector`**? Yes!

```
int main()
{
  double d1[5] = {1, 2, 3, 4, 5};
  const MyVector a1(5, d1);
  cout << a1[3] << endl; // endl is a newline object
  a1[1] = 4;

  return 0;
}
```

Overloading the indexing operator

- Let's overload `[]`:

```
class MyVector
{
  // ...
  double operator[](int i) const;
};
```

```
double MyVector::operator[](int i) const
{
  if(i < 0 || i >= n)
    exit(1); // terminate the program!
             // required <cstdlib>
  return m[i];
}
```

  - **`exit(1)`** terminates the program by sending 1 to the operating system.
  - **`return 0`** in the main function terminates the program by sending 0.
  - 0: Normal termination. Other numbers: different errors.

More are needed for `[]`

- Compiling the program with the main function below results in an error!

```
int main()
{
  double d1[5] = {1, 2, 3, 4, 5};
  MyVector a1(5, d1); // non-const
  cout << a1[3] << endl; // good
  a1[1] = 4; // error!
  return 0;
}
```

- Error: **`a1[1]`** is just a **literal**, not a variable.
  - A literal cannot be put at the LHS in an assignment operation!
  - Just like **`3 = 5`** results in an error.

Another overloaded `[]`

- Let's overload `[]` into another version:

```
class MyVector
{
  // ...
  double operator[](int i) const;
  double& operator[](int i);
};
```

```
double MyVector::operator[](int i) const
{
  if(i < 0 || i >= n)
    exit(1);
  return m[i];
}
double& MyVector::operator[](int i)
{
  if(i < 0 || i >= n) // same
    exit(1);          // implementation!
  return m[i];
}
```

- The second implementation returns a **reference** of a member variable.
  - Modifying that reference modifies the variable.

Two different `[]`

- Now the program runs successfully!

```
int main()
{
  double d1[5] = {1, 2, 3, 4, 5};
  MyVector a1(5, d1);
  cout << a1[1] << endl; // 2
  a1[1] = 4; // good
  cout << a1[1] << endl; // 4

  return 0;
}
```

```
double MyVector::operator[](int i) const
{
  if(i < 0 || i >= n)
    exit(1);
  return m[i];
}
double& MyVector::operator[](int i)
{
  if(i < 0 || i >= n)
    exit(1);
  return m[i];
}
```

- There is one last question:
  - Which `[]` is invoked?

Invoking the two [ ]

- The **const** after the function prototype is the key.

```
class MyVector
{
  // ...
  double operator[](int i) const;
  double& operator[](int i);
};
```

- If there are both a constant and a non-constant version:
  – A constant function is invoked by a constant object.
  – A non-constant function is invoked by a non-constant object.
- If there is only a non-constant instance function:
  – A constant object cannot invoke it.

Outline

- Motivations and prerequisites
- Overloading comparison and indexing operators
- **Overloading assignment and self-assignment operators**
- Overloading addition operators

Operations that modify the object

- Some operations do not modify the calling object.
  - E.g., comparisons and indexing.
- Some operations modify the calling object.
  - E.g., assignments and self-assignments.
  - Let's overload the assignment operator **=** first.
- What do we expect?

```
int main()
{
  double d1[3] = {1, 2, 3};
  double d2[4] = {1, 2, 3, 4};
  MyVector a1(3, d1);
  MyVector a2(4, d2);

  a2.print();
  a2 = a1; // assignment
  a2.print();

  return 0;
}
```

Default assignment operator

- In fact, the assignment operator has been overloaded!
  - The compiler adds a **default assignment operator** into each class.
  - It simply **copies each instance variable** to its corresponding one.
  - Just like the default copy constructor.
- What's wrong if we use the default assignment operator?

```
int main()
{
  double d1[3] = {1, 2, 3};
  double d2[4] = {1, 2, 3, 4};
  MyVector a1(3, d1);
  MyVector a2(4, d2);

  a2.print();
  a2 = a1; // dangerous
  a2.print();

  return 0;
}
```

Default assignment operator

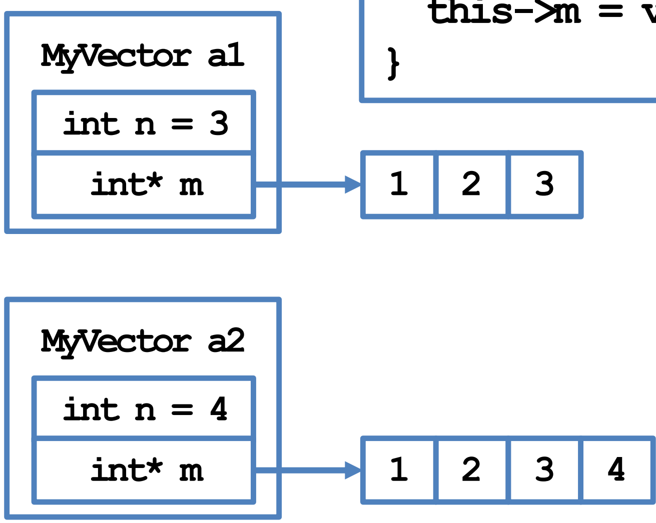
```
MyVector& MyVector::operator=
  (const MyVector& v)
{ // the default one
  this->n = v.n;
  this->m = v.m;
}
```

```
int main()
{
  double d1[3] = {1, 2, 3};
  double d2[4] = {1, 2, 3, 4};
  MyVector a1(3, d1); // (1)
  MyVector a2(4, d2); // (2)

  a1.print();
  a1 = a2; // (3): dangerous
  a1.print();

  return 0;
}
```

(1)



(2)

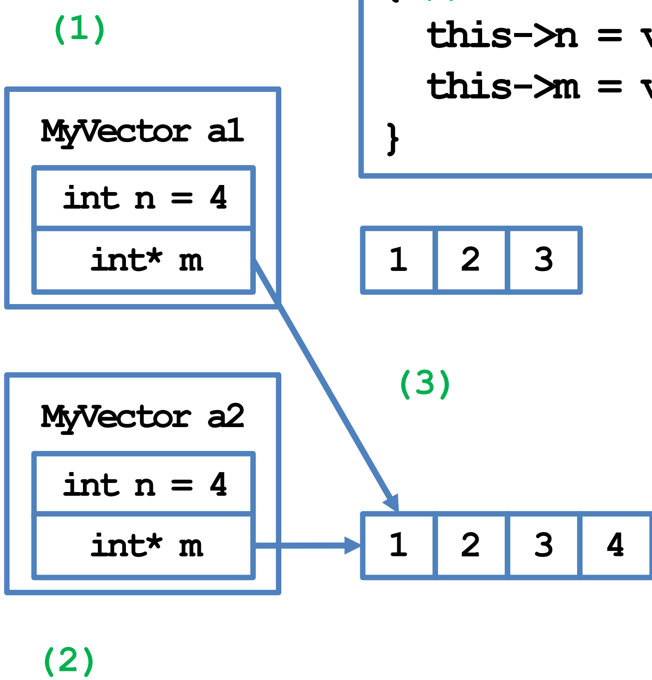
Default assignment operator

```
MyVector& MyVector::operator=
    (const MyVector& v)
{ // the default one
    this->n = v.n;
    this->m = v.m;
}
```

```
int main()
{
    double d1[3] = {1, 2, 3};
    double d2[4] = {1, 2, 3, 4};
    MyVector a1(3, d1); // (1)
    MyVector a2(4, d2); // (2)

    a1.print();
    a1 = a2; // (3): dangerous
    a1.print();

    return 0;
}
```



Overloading the assignment operator

- Just like the copy constructor, the assignment operator should be manually overloaded when there are pointers in a class.
- Our first implementation:

```
class MyVector
{
  // ...
  void operator=(const MyVector& v);
};
```

- If one execute **`a1 = a1`**, we need to copy all elements while it is not needed. How to save time?

```
void MyVector::operator=(const MyVector& v)
{
  if(this->n != v.n)
  {
    delete [] this->m;
    this->n = v.n;
    this->m = new double[this->n];
  }
  for(int i = 0; i < n; i++)
    this->m[i] = v.m[i];
}
```

Overloading the assignment operator

- Our second implementation:

```
class MyVector
{
  // ...
  void operator=(const MyVector& v);
};
```

- This does not allow one to do **a1 = a2 = a3**. How to make this possible?

```
void MyVector::operator=(const MyVector& v)
{
  if(this != &v)
  {
    if(this->n != v.n)
    {
      delete [] this->m;
      this->n = v.n;
      this->m = new double[this->n];
    }
    for(int i = 0; i < n; i++)
      this->m[i] = v.m[i];
  }
}
```

Overloading the assignment operator

- Our third implementation:

```
class MyVector
{
  // ...
  MyVector& operator=
     (const MyVector& v);
};
```

- If we want to prevent `(a1 = a2) = a3`, we may instead return `const MyVector&`.

```
MyVector& MyVector::operator=
   (const MyVector& v)
{
  if(this != &v)
  {
    if(this->n != v.n)
    {
      delete [] this->m;
      this->n = v.n;
      this->m = new double[this->n];
    }
    for(int i = 0; i < n; i++)
      this->m[i] = v.m[i];
  }
  return *this;
}
```

Preventing assignments and copying

- In some cases, we **disallow** assignments between objects of a certain class.
  - To do so, overload the assignment operator as a **private** member.
- In some cases, we disallow creating an object by **copying** another object.
  - To do so, implement the copy constructor as a **private** member.
- The copy constructor, assignment operator, and destructor form a group.
  - If there is no pointer, **none** of them is needed.
  - If there is a pointer, **all** of them are needed.

Self-assignment operators

- For vectors, it is often to do arithmetic and assignments.
  - Given vectors u and v of the same dimension, the operation u += v makes u_i become $u_i + v_i$ for all i.
- Let's overload **+=**:
- Why returning **const MyVector&**?
  - Returning **MyVector&** allows **(a1 += a3)[i]**.
  - Returning **const MyVector&** disallows **(a1 += a3) = a2**.

```
class MyVector
{
  // ...
  const MyVector& operator+=
    (const MyVector& v);
};
const MyVector& MyVector::operator+=
  (const MyVector& v)
{
  if(this->n == v.n)
  {
    for(int i = 0; i < n; i++)
      this->m[i] += v.m[i];
  }
  return *this;
}
```

Outline

- Motivations and prerequisites
- Overloading comparison and indexing operators
- Overloading assignment and self-assignment operators
- **Overloading addition operators**

Arithmetic operators

- Overloading an arithmetic operator is not hard.
- Consider the addition operator `+` as an example.
  - Take `const MyVector&` as a parameter.
  - Add each pair of elements one by one.
  - Do not modify the calling and parameter objects.
  - Return `const MyVector` to allow `a1 + a2 + a3` but disallow `(a1 + a2) = a3`.

Overloading the addition operator

- Let's try to do it.

```
class MyVector
{
  // ...
  const MyVector operator+(const MyVector& v);
};
const MyVector MyVector::operator+(const MyVector& v)
{
  MyVector sum(*this); // creating a local variable
  sum += v; // using the overloaded +=
  return sum;
}
```

- Why not returning **`const MyVector&`**?
  - Hint: What will happen to **`sum`** after the function call is finished?

Overloading the addition operator

- We may overload it for another parameter type:

```
int main()
{
  double d1[5] = {1, 2, 3};
  MyVector a1(3, d1);
  MyVector a2(3, d1);

  a1 = a1 + a2; // good
  a1.print();
  a1 = a2 + 4.2; // good
  a1.print();

  return 0;
}
```

```
class MyVector
{
  // ...
  const MyVector operator+(double d);
};
const MyVector MyVector::operator+(double d)
{
  MyVector sum(*this);
  for(int i = 0; i < n; i++)
    sum[i] += d;
  return sum;
}
```

Instance function vs. global function

- One last issue: addition is **commutative**, but the program below does not run!

```
int main()
{
  double d1[5] = {1, 2, 3, 4, 5};
  MyVector a1(5, d1);
  a1 = 4.2 + a1; // bad!
  a1.print();

  return 0;
}
```

- We cannot let a double variable invoke our "instance function **operator+**".
- We should overload **+** as a **global function**.

A global-function version

- To overload **+** as global functions, we need to handle the three combinations:

```
const MyVector operator+
  (const MyVector& v, double d)
{ // need to be a friend of MyVector
  MyVector sum(v);
  for(int i = 0; i < v.n; i++)
    sum[i] += d; // pairwise addition
  return sum;
}
const MyVector operator+
  (double d, const MyVector& v)
{
  return v + d; // using the previous definition
}
```

```
const MyVector operator+
  (const MyVector& v1,
   const MyVector& v2)
{
  MyVector sum(v1);
  sum += v2; // using +=
  return sum;
}
```

A global-function version

- Now all kinds of addition may be performed:

```
int main()
{
  double d1[5] = {1, 2, 3, 4, 5};
  MyVector a1(5, d1);
  MyVector a3(a1);

  a3 = 3 + a1 + 4 + a3;
  a3.print();

  return 0;
}
```

- Each operator needs a separate consideration.