Contemporary C++: *Learning Modern C++ in a Modern Way*

الماس فناوری ابری پاسارگاد– آلفا

مدرس: سعید امراللهی بیوکی

Agenda 10/24

Session 10. Go deep on classes: Copy operations, overloaded operators, class invariants and more

- Special member functions: copy constructors and copy assignment operators
- Defaulted and deleted member functions
- Why operator overloading?
- Operator functions
- Operators and User-defined types
- Overloaded operators: complex numbers
- Vector element access: the subscription operator ()
- Q&A

150 min (incl. Q & A)

Concrete classes: copy operations

Language-technical rule

- a concrete type should be resemble a built-in type.

Provide as good support for user-defined types as for built-in types.

```
void f(int a, int& b)
{
  int i;
  int j = a;
  i = b;
  int* pi = &i;
  int* p = new int{42};
  *pp = 43;
  // …
  delete p;
}
```

```
class Date {
     // …
};
void f(Date my_birthday, Date& tomorrow)
{
  Date d; // calling default constructor
  Date copy_of_d = my_birthday; // calling copy constructor
  d = tomorrow; // calling copy assignment operator
  Date* p = &d; // take the address of d
  Date* pd = new Date{1, 9, 2023}; // make -unnamed- date object in heap
  pd->add_year(1);
  // …
  delete pp; // release the memory allocated by pd point to.
}
```

Copying objects

- The default behavior of copy operations is *member-wise* copy.
- By default, the copy of a class object is a copy of each member.

```
Date today(11, 9, 2000); // default ctor
Date d = today; // initialization: d.Day = 11, d.Month = 9, d.Year = 2000
d.add_day(1); // d.Year = 2001
Date tomorrow; // default ctor
tomorrow = d; // assignment: d.Day = 11, d.Month = 9, d.Year = 2001
```

- The default behavior: Copy all the data members.
- The behavior of generated default copy operations of Date, Point are correct.

Concrete Classes Copy Operations

Prog.

Copy operations: copy constructor and assignment operator

Copy operations = copy constructor + copy assignment operator

Copy constructor

Thing t = x; **// Initialization (new Thing created)**

t = x; **// Assignment (value of existing Thing changed)**

Copy assignment operator

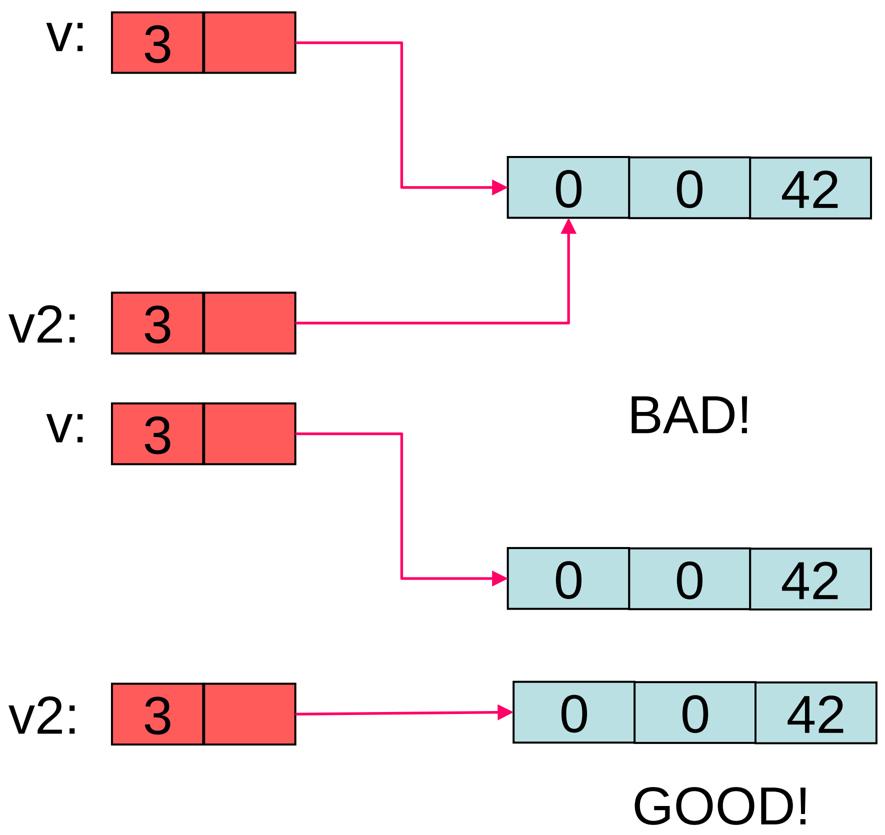
Implicitly generated

Implicitly generated

```
class X {
public:
   X(const X&); // Copy ctor
   X& operator=(const X&); // assignment operator
};
inline X::X(const X& rhs) { /* member-wise copy */ }
inline X& X::operator=(const X& rhs)
{
     /* member-wise assignment */
     return *this;
}
```

Copy constructor: vector

```
Void f()
{
     Vector v(3);
     v.set(2, 42);
     Vector v2 = v; // calling copy ctor
}
```



- v2 doesn't have a copy of v's elements, but it shares v's elements.
- Double deletion

1

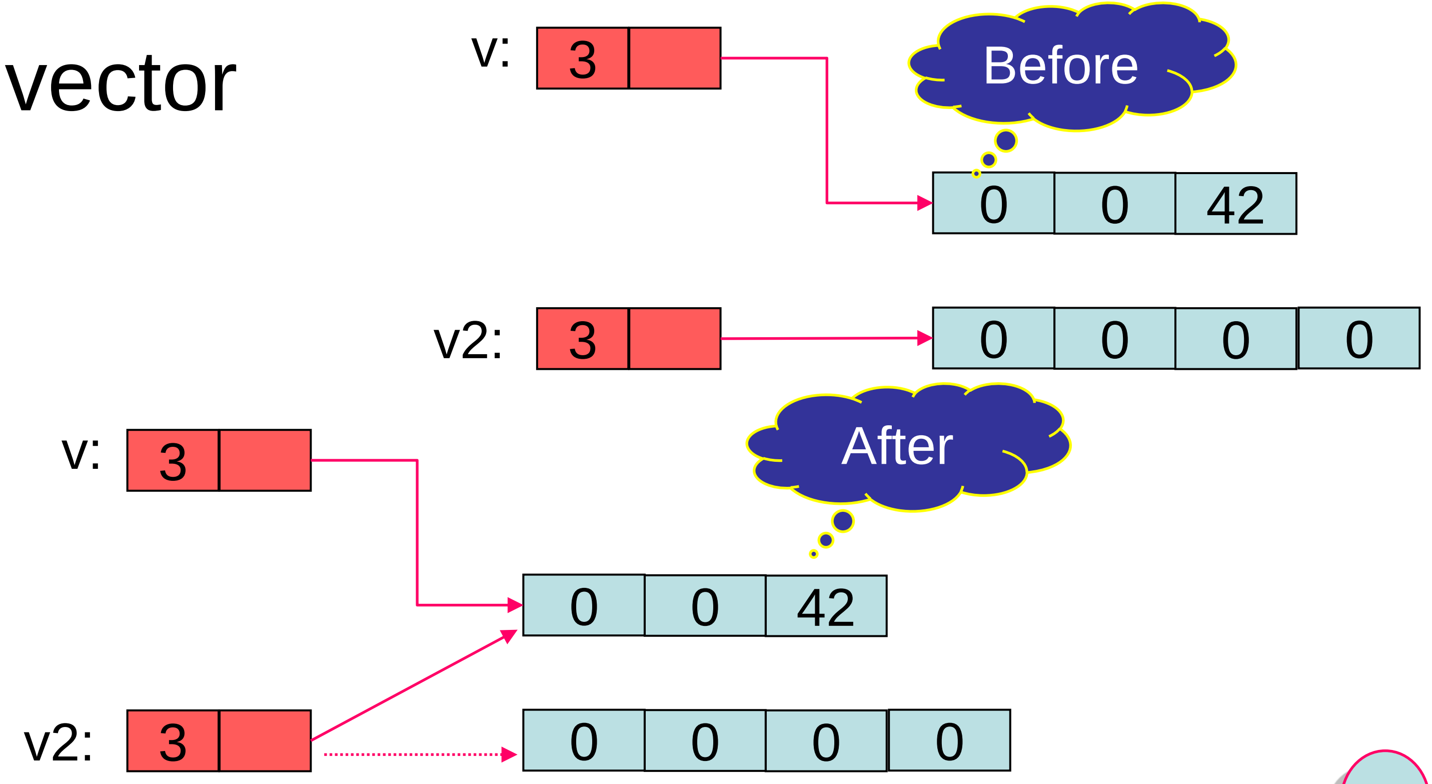
```
// vector.h
class Vector {
     int sz;
     int* elem;
public:
     Vector(const Vector&);
     // …
};
```

2

```
// vector.cpp
Vector::Vector(const Vector& v)  :
     // allocate elements, then initialize them by copying
     elem{new int[v.sz]), ss{v.sz}
{
     for (int i =0; i < v.sz; ++i)
          elem[sz] = v[sz];
}
```

Copy assignment operator: vector

```
Void f()
{
     Vector v(3);
     v.set(2, 42);
     Vector v2(4); // calling copy ctor
     v2 = v;
     // …
}
```



- v2 doesn’t have a copy of v’s elements, but it shares v’s elements.
- Double deletion + memory leak

1

```
// vector.h
class Vector {
     int sz;
     int* elem;
public:
     Vector& operator=(const Vector&);
     // …
};
```

2

```
// vector.cpp
Vector& Vector::operator=(const Vector& v)
{
     int* p = new int[v.sz];
     for (int i =0; i < v.sz; ++i)
          elem[sz] = v[sz];
     delete [] elem;
     elem = p;
     sz = v.sz;
     return *this;
}
```

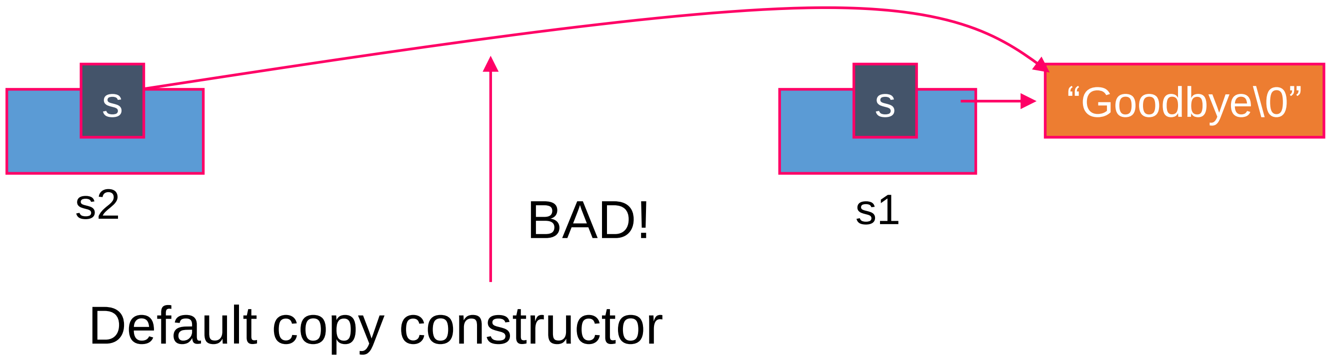
Copy operations: the String class

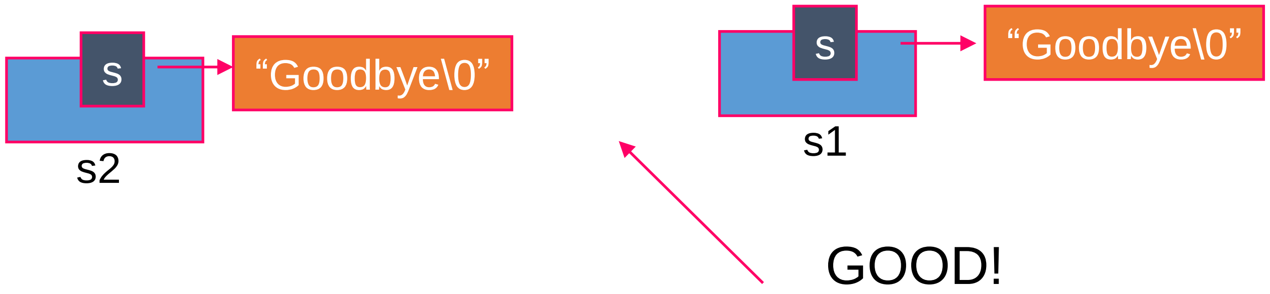
- String: copy constructor

```
String s1(“Goodbye”);
String s2 = s1; // calling copy constructor
```

```
// String.h
class String {
public:
  String(const String&); // Copy Ctor
  // …
};
```

```
#include <cstring>
String::String(const String& str)
{
  s = new char[strlen(str.s) + 1];
  strcpy(s, str.s);
}
```

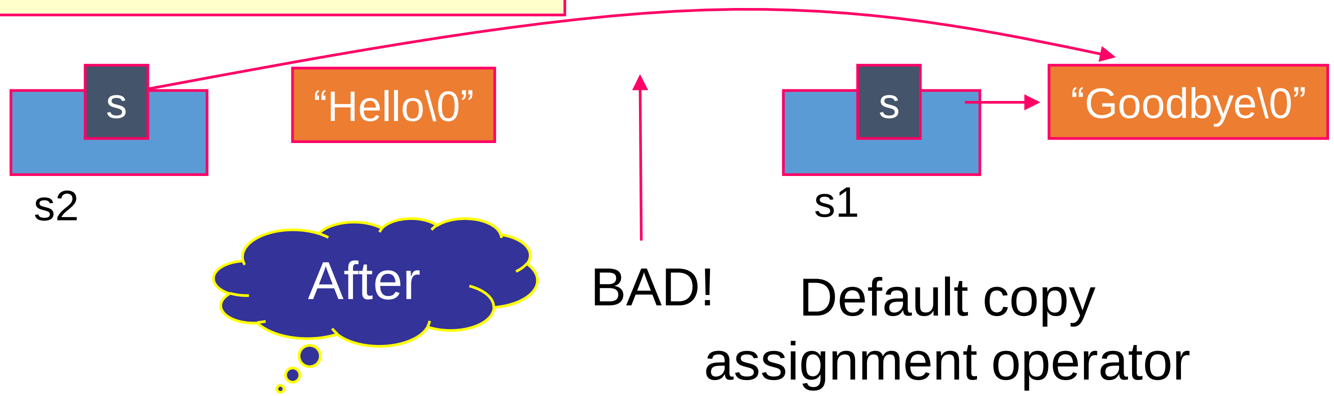




Copy operations: the String class cont.

- String: assignment operator

```
String s1("Goodbye");
String s2("Hello");
s2 = s1; // calling copy assignment operator
```



Default copy assignment operator

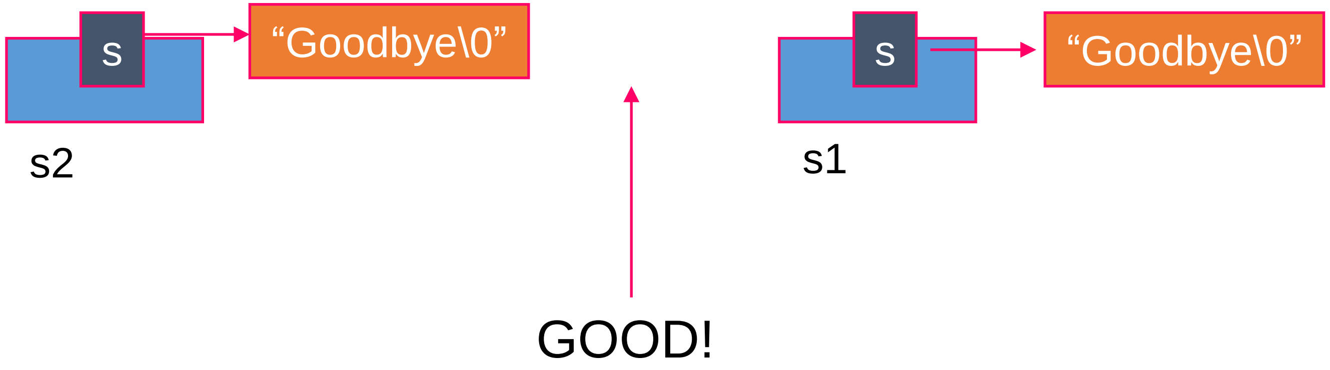
- Overriding generated default copy assignment operator- 1st try
- Override: To re-implement

```
// String.h
class String {
public:
  void operator=(const String&); // copy assignment operator
};
```

Overloaded operators

```
// Naïve and Inkorect
void String::operator=(const String& str)
{
  delete [] s;
  s = new char[strlen(str.s) + 1];
  strcpy(s, str.s);
}
```

Copy operations: the String class cont.



- Handling self-assignment
- Handling multiple and chaining assignments
- Overriding generated default copy assignment operator- 2nd try

```
String s;
s = s; // doesn't handle self assignment
String s1, s2, s3;
String s4("ABC");
s1 = s2 = s3 = s4; // doesn't handle multiple assignment
```

2

```
// Good
String& String::operator=(const String& str)
{
  if (this != &str) { // self assignment
   delete [] s;
   s = new char[strlen(str.s) + 1];
   strcpy(s, str.s);
  }
  return *this; // multiple assignment
}
```

Defaulted and deleted functions

- Example: C++ Common idiom: Prohibiting copy operations.

```
// C++98
class X {
   X(const X&);
   X& operator=(const X&);
    // …
};
X one; // OK: calling default ctor
X clone = one; // error: private copy ctor
X another_one;
another_one = one; // error: private copy assignment operator
```

- Prohibit copy operations: an example

```
// C++98
class EEPROM_wrapper {
     // …
private:
    // prohibit copy operations
    EEPROM_wrapper(const EEPROM_wrapper&);
    EEPROM_wrapper& operator=(const EEPROM_wrapper&);
};
```

Defaulted and deleted functions

- C++11 is more explicit about prohibiting a function:
- Reuse of a keyword: delete

Explicitly deleted functions ←

```
// C++11
class X { // can't copy it
   X(const X&) = delete;
   X& operator=(const X&) = delete;
    // …
};
```

- Deleted function has deleted definition.
- The "delete" mechanism can be used for any function.

```
struct Z {
     Z(long long); // can initialize with an long long
     Z(long) = delete; // but not anything less
};
```

- Example from standard library

```
class thread {
   thread(const thread&) = delete;
   thread& operator=(const thread&) = delete;
};
```

Defaulted and deleted functions cont.

- We can be explicit about default behavior.

```
class Point {
  int x, y;
};
```

C++98 compiler coverts to

```
class Point {
  int x, y;
public:
  // default behavior of special member functions
  Point() {} // 'default' default ctor
  Point(const Point& RHS) : x(RHS.x), y(RHS.y) {} // 'default' copy ctor
  // 'default' copy assignment operator
  Point& operator=(const Point& RHS) { x = RHS.x; Y = RHS.y; return *this; }
  ~Point() {} // 'default' dtor
};
```

- Being explicit about the default is redundant. However, comments about copy operations and (worse) a user explicitly defining copy operations meant to give the default behavior are not uncommon. Leaving it to the compiler to implement the default behavior is simpler, less error-prone, and often leads to better object code.

```cpp
// C++11
class X { // can’t copy it
   X(const X&) = default;
   X& operator=(const X&) = default;
    // …
};
```

- The "default" mechanism can be used for any function that has a default -> Special member functions

Overloading

Overloading
- Function name overloading
- Operator overloading

Overloading
- Function name overloading
- Operator overloading

• Overloading: having more than one *function* with the same name in the same scope or having more than one *operator* with the same in the same scope. http://www.stroustrup.com/glossary.html#Goverloading

Operator function: An example

Operator function: An example

- We don’t need operator overloading.

```
F = M * A // Assign(F, Multiply(M, A))
```

Operator function: An example

- We don't need operator overloading.

```
F = M * A // Assign(F, Multiply(M, A))
```

- Polynomials:

p_1: $-x^4 + 3x^2 + 4x - 1$

p_2: $x^5 + 1$ p_3: $2x + 1$

$p = p_1 + p_2 * p_3 / p_1;$

Operator function: An example

- We don't need operator overloading.

```
F = M * A // Assign(F, Multiply(M, A))
```

p1: $-x^4 + 3x^2 + 4x - 1$

p2: $x^5 + 1$ p3: $2x + 1$

$p = p_1 + p_2 * p_3 / p_1;$

- Polynomials:

1.

```
class Polynomial {
public:
   // ctor(s)
   // …
   // member functions
   Polynomial& add(const Polynomial&);
   Polynomial& sub(const Polynomial&);
   Polynomial& mul(const Polynomial&);
   Polynomial& div(const Polynomial&);
   // …
};
```

ALPHA آلفا

Operator function: An example

- We don't need operator overloading.

```
F = M * A // Assign(F, Multiply(M, A))
```

p1: $-x^4 + 3x^2 + 4x - 1$

p2: $x^5 + 1$

p3: $2x + 1$

- Polynomials:

1.

```
class Polynomial {
public:
   // ctor(s)
   // …
   // member functions
   Polynomial& add(const Polynomial&);
   Polynomial& sub(const Polynomial&);
   Polynomial& mul(const Polynomial&);
   Polynomial& div(const Polynomial&);
   // …
};
```

$p = p_1 + p_2 * p_3 / p_1;$

2.

```
class Polynomial {

public:
   // operator functions
   Polynomial& operator+=(const Polynomial&);
   Polynomial& operator-=(const Polynomial&);
   Polynomial& operator*=(const Polynomial&);
   Polynomial& operator/=(const Polynomial&);
   // …
};

Polynomial operator+(const Polynomial&, const Polynomial&);
Polynomial operator-(const Polynomial&, const Polynomial&);
Polynomial operator*(const Polynomial&, const Polynomial&);
Polynomial operator/(const Polynomial&, const Polynomial&);
```

Operator function: An example

- We don't need operator overloading.

```
F = M * A // Assign(F, Multiply(M, A))
```

- Polynomials:

p1: $-x^4+3x^2+4x-1$

p2: x^5+1 p3: $2x+1$

$p = p_1 + p_2 * p_3 / p_1;$

1.

```
class Polynomial {
public:
   // ctor(s)
   // …
   // member functions
   Polynomial& add(const Polynomial&);
   Polynomial& sub(const Polynomial&);
   Polynomial& mul(const Polynomial&);
   Polynomial& div(const Polynomial&);
   // …
};
```

- Overloaded operators = Operator functions

2.

```
class Polynomial {

public:
   // operator functions
   Polynomial& operator+=(const Polynomial&);
   Polynomial& operator-=(const Polynomial&);
   Polynomial& operator*=(const Polynomial&);
   Polynomial& operator/=(const Polynomial&);
   // …
};

Polynomial operator+(const Polynomial&, const Polynomial&);
Polynomial operator-(const Polynomial&, const Polynomial&);
Polynomial operator*(const Polynomial&, const Polynomial&);
Polynomial operator/(const Polynomial&, const Polynomial&);
```

Operator functions

Operator functions

Overloaded operators advantages

Overloaded operators advantages

- Without overloaded operators

```
Polynomial f(Polynomial p1, Polynomial p2, Polynomial p3) {
  return p1.Add((p2.Mul(p3)).Div(p1));
}
```

Overloaded operators advantages

- Without overloaded operators

```
Polynomial f(Polynomial p1, Polynomial p2, Polynomial p3) {
  return p1.Add((p2.Mul(p3)).Div(p1));
}
```

Solution domain terminology

Overloaded operators advantages

- Without overloaded operators

```
Polynomial f(Polynomial p1, Polynomial p2, Polynomial p3) {
  return p1.Add((p2.Mul(p3)).Div(p1));
}
```

Solution domain terminology

- With overloaded operators

```
Polynomial f(Polynomial p1, Polynomial p2, Polynomial p3) {
  return p1 + p2 * p3 / p1;
}
```

Overloaded operators advantages

- Without overloaded operators

```
Polynomial f(Polynomial p1, Polynomial p2, Polynomial p3) {
  return p1.Add((p2.Mul(p3)).Div(p1));
}
```

Solution domain terminology

- With overloaded operators

```
Polynomial f(Polynomial p1, Polynomial p2, Polynomial p3) {
  return p1 + p2 * p3 / p1;
}
```

Problem domain terminology

Overloaded operators advantages

- Without overloaded operators

```
Polynomial f(Polynomial p1, Polynomial p2, Polynomial p3) {
  return p1.Add((p2.Mul(p3)).Div(p1));
}
```

Solution domain terminology

- With overloaded operators

```
Polynomial f(Polynomial p1, Polynomial p2, Polynomial p3) {
  return p1 + p2 * p3 / p1;
}
```

Problem domain terminology

- Polynomial interface communicates with polynomial abstraction.

Overloaded operators advantages

- Without overloaded operators

```
Polynomial f(Polynomial p1, Polynomial p2, Polynomial p3) {
  return p1.Add((p2.Mul(p3)).Div(p1));
}
```

Solution domain terminology

- With overloaded operators

```
Polynomial f(Polynomial p1, Polynomial p2, Polynomial p3) {
  return p1 + p2 * p3 / p1;
}
```

Problem domain terminology

- Polynomial interface communicates with polynomial abstraction.
- Polynomial with overloaded operators are more readable and maintainable.

Overloaded operators advantages

- Without overloaded operators

```
Polynomial f(Polynomial p1, Polynomial p2, Polynomial p3) {
  return p1.Add((p2.Mul(p3)).Div(p1));
}
```

Solution domain terminology

- With overloaded operators

```
Polynomial f(Polynomial p1, Polynomial p2, Polynomial p3) {
  return p1 + p2 * p3 / p1;
}
```

Problem domain terminology

- Polynomial interface communicates with polynomial abstraction.
- Polynomial with overloaded operators are more readable and maintainable.
- Polynomial with overloaded operators are more efficient.

Operator overloading

Operator overloading

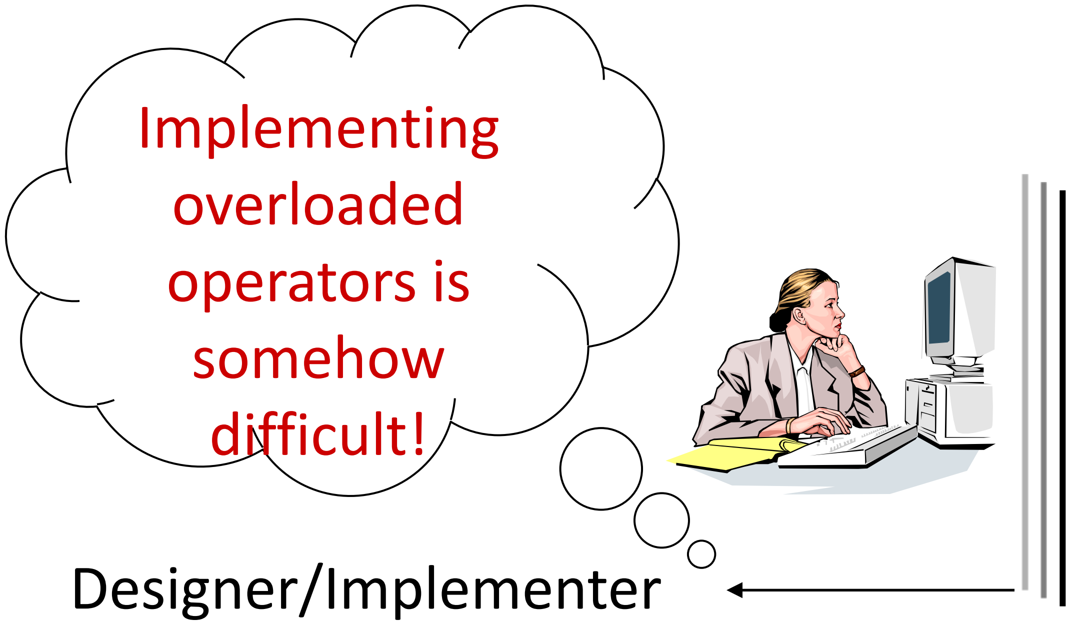
- String manipulation, Matrix algebra, Tensor algebra, Electric circuits, Electronics, Complex numbers arithmetic, Numeric methods, …

Operator overloading

- String manipulation, Matrix algebra, Tensor algebra, Electric circuits, Complex numbers arithmetic, Numeric methods, …

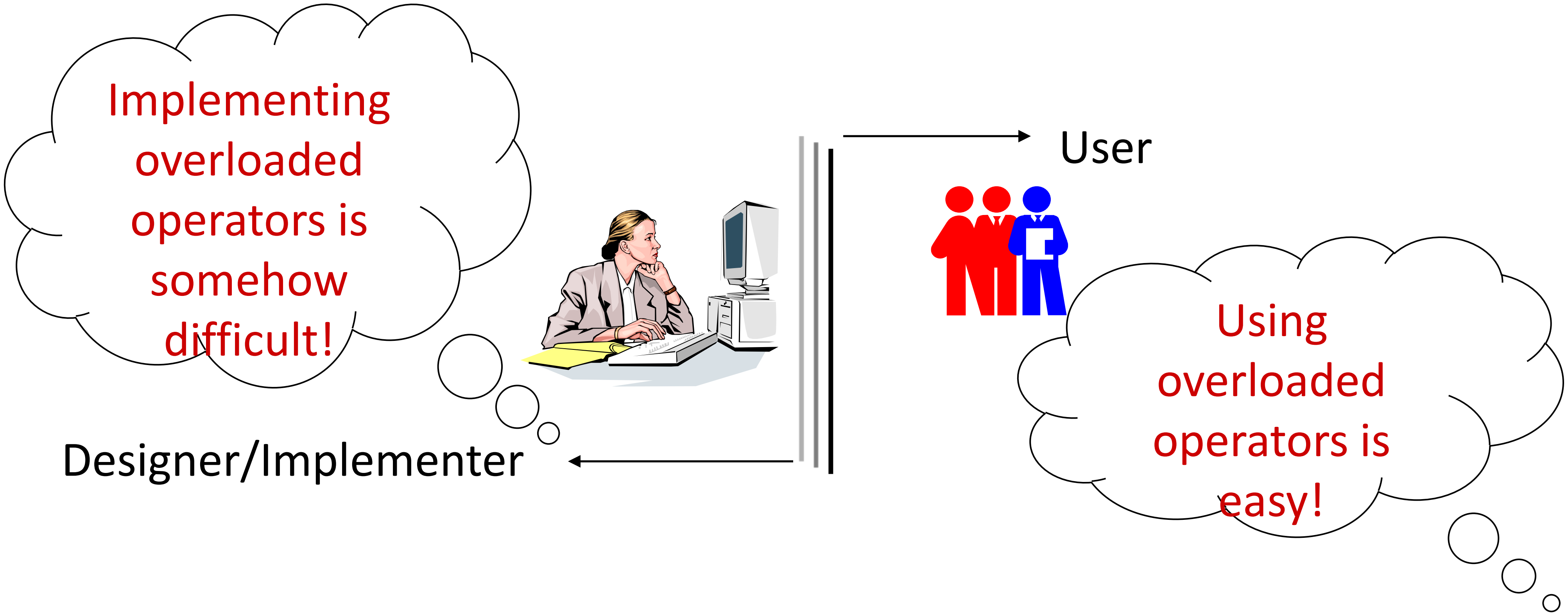
Operator overloading

- String manipulation, Matrix algebra, Tensor algebra, Electric circuits, Complex numbers arithmetic, Numeric methods, …



Operator overloading

- String manipulation, Matrix algebra, Tensor algebra, Electric circuits, Complex numbers arithmetic, Numeric methods, …



Operator functions

- operator function: function defining one of the standard operators; e.g. operator+().
- The following operators can be overloaded:

C++98

+	-	*	/	%	^
&	|	~	!	=	<
>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&
||	++	--	->*	,	->
[]	()	new	new[]	delete	delete[]

Subscription

Function call

More than one token

- The name of an operator function is the keyword `operator` followed by the operator itself. They are exactly like ordinary functions.
  - Examples:

```
String& operator=(const String&); // copy assignment operator
iostream& operator<<(iostream&, int); // put to or extractor operator
BigInt& operator+(const BigInt&, const BigInt&); // add operator for class BigInt
bool operator==(const Date&, const Date&); // equality operator for class Date
```

General rules

- The following operators cannot be defined by user:

`::` (scope resolution),
`.` (member selection),
`.*` (member selection through pointer to function)
`?:`

- At least one argument must be user-defined types: class, enumeration, …

- The usual precedence rules hold. It is impossible to change the precedence and associativity of existing operators.

- No new operators may be defined.

```
pow() vs. **
```

- It is not possible to change the predefined meanings of *assignment*, *address-of* and *sequencing* operators.

- Member operators must be non-static.

- operator=, operator[], operator(), and operator -> must be member functions.

```
class X {
public:
  X& operator=(const X&);
  X& operator&() const;
  void operator,();
};
```

General rules cont.

- Built-in operators equivalence for built-in type are not applied to user-defined types.

```
int a = 1;
a++; // means a += 1 also means a = a + 1;
```

- For user-defined types, programmer should define all needed operators:

```
class X {
public:
   X& operator++(X);
   X& operator+=(X);
   X& operator=(const X&);
   X& operator+(X);
   // …
};
```

Binary and unary operators

- A binary operator can be defined by either
  - a *non-static member function* taking one argument or
  - a *nonmember function* taking two arguments.

For any binary operator **@**, **aa@bb** can be interpreted as either **aa.operator@(bb)** or **operator@(aa, bb)**.

- A unary operator, whether prefix or postfix, can be defined by either
  - a *non-static member function* taking no arguments or
  - a *nonmember function* taking one argument.

For any prefix unary operator `@`, `@aa` can be interpreted as either `aa.operator@()` or `operator@(aa)`.

For any postfix unary operator `@`, `aa@` can be interpreted as either `aa.operator@(int)` or `operator@(aa ,int)`.

- Overload resolution: first member candidates, then non-member candidates.

```
// Binary operator
class X {
public:
  void operator+(X); (1)
};
void operator+(X, X); (2)
```

```
// Unary operators
class X {
public:
  void operator++(); (1)
};
void operator++(X); (2)
```

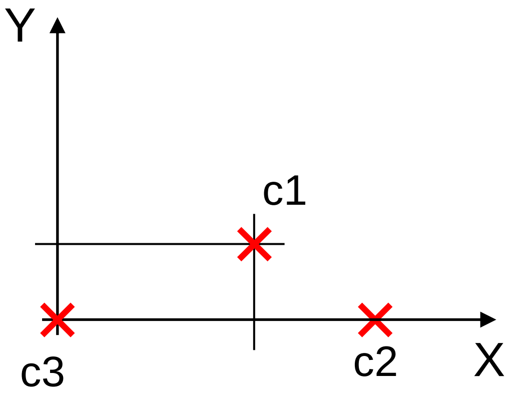
an example from standard library: Complex numbers

an example from standard library: Complex numbers

Complex number = a + i*b

a: real part

b: imaginary part

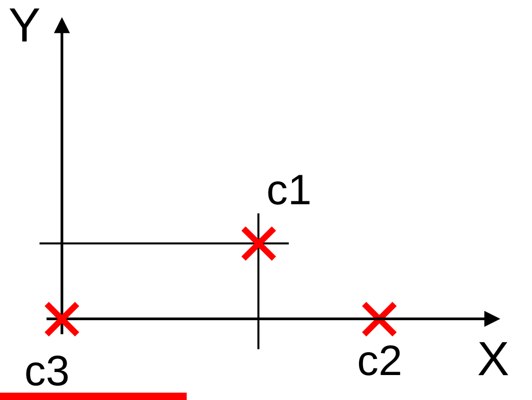


an example from standard library: Complex numbers

Complex number = a + i*b

a: real part

b: imaginary part



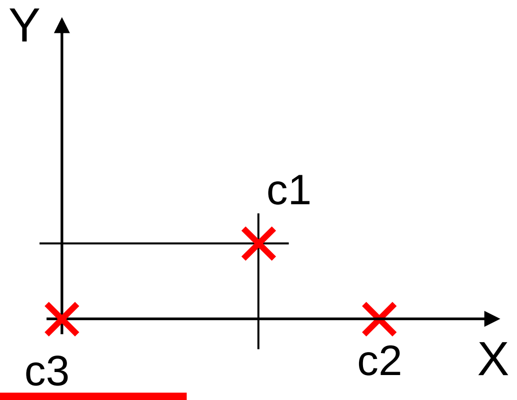

```
// very simplified
class Complex { // complex number concept
public: // ctors
  Complex(double r, double i) { re = r; im = i; }
  Complex(double r) { re = r; im = 0.0; }
  Complex() { re = im = 0.0; }
  // overloaded operators: just an example
  Complex& operator+=(const Complex&); // Complex Add(Complex)!
  Complex& operator*=(const Complex&); // Complex Mul(Complex)!
  double real() const { return re; }
  void real(double re_) { re = re_; }
  double imag() const { return im; }
  void imag(double im_) { im = im_; }
private: // representation
  double re, im;
};
```

an example from standard library: Complex numbers

Complex number = a + i*b

a: real part

b: imaginary part



Overloaded operators

Overloaded functions

```
// very simplified
class Complex { // complex number concept
public: // ctors
  Complex(double r, double i) { re = r; im = i; }
  Complex(double r) { re = r; im = 0.0; }
  Complex() { re = im = 0.0; }
  // overloaded operators: just an example
  Complex& operator+=(const Complex&); // Complex Add(Complex)!
  Complex& operator*=(const Complex&); // Complex Mul(Complex)!
  double real() const { return re; }
  void real(double re_) { re = re_; }
  double imag() const { return im; }
  void imag(double im_) { im = im_; }
private: // representation
  double re, im;
};
```

Complex numbers cont.

Complex numbers cont.

1

```
Complex& Complex::operator+=(const Complex& c)
{
    re += c.re;
    im += c.im;
    return *this;
}

Complex& Complex::operator*=(const Complex& c)
{
    re = re * c.re - im * c.im;
    im = re * c.im + im * c.re;

    return cc;
}
```

Complex numbers cont.

1

```
Complex& Complex::operator+=(const Complex& c)
{
    re += c.re;
    im += c.im;
    return *this;
}

Complex& Complex::operator*=(const Complex& c)
{
    re = re * c.re - im * c.im;
    im = re * c.im + im * c.re;

    return cc;
}
```

2

```
void f()
{
 Complex a = Complex{1, 3.1};
 Complex b = Complex{1.2, 2};
 Complex c = b;
 a = b + c; // a = b.operator+(c)
 b = b + c * a; // b = b.operator+(c.operator*(a))
 c = a * b + Complex{1,2};
}
```

Complex numbers cont.

1

```
Complex& Complex::operator+=(const Complex& c)
{
    re += c.re;
    im += c.im;
    return *this;
}

Complex& Complex::operator*=(const Complex& c)
{
    re = re * c.re - im * c.im;
    im = re * c.im + im * c.re;

    return cc;
}
```

2

```
void f()
{
 Complex a = Complex(1, 3.1);
 Complex b = Complex(1.2, 2);
 Complex c = b;
 a = b + c; // a = b.operator+(c)
 b = b + c * a; // b = b.operator+(c.operator*(a))
 c = a * b + Complex(1,2);
}
```

3

```
void f(Complex a, Complex b)
{
  Complex c = a + b; // shorthand
  Complex d = a.operator+(b); // explicit call
}
```

Complex numbers cont.

1

```
Complex& Complex::operator+=(const Complex& c)
{
     re += c.re;
     im += c.im;
     return *this;
}

Complex& Complex::operator*=(const Complex& c)
{
     re = re * c.re - im * c.im;
     im = re * c.im + im * c.re;

     return cc;
}
```

2

```
void f()
{
 Complex a = Complex(1, 3.1);
 Complex b = Complex(1.2, 2);
 Complex c = b;
 a = b + c; // a = b.operator+(c)
 b = b + c * a; // b = b.operator+(c.operator*(a))
 c = a * b + Complex(1,2);
}
```

3

```
void f(Complex a, Complex b)
{
  Complex c = a + b; // shorthand
  Complex d = a.operator+(b); // explicit call
}
```

- Operator overloading is like "*syntactic sugar*".

Operator functions: some examples

- Enumeration:

```
enum Day { sun, mon, tue, wed, thu, fri, sat };
Day& operator++(Day& d)
{
 return d = (sat == d) ? sun: Day(d+1);
}
```

```
enum Season { winter, spring, summer, fall };
void operator++(Season& s)
{
  switch (s) {
    case winter: s = spring; break;
    case spring: s = summer; break;
    case summer: s = fall;   break;
    case fall:   s = winter; break;
    default: break;
  }
}
```

Operator functions: class Date

- Equality operator

```
inline bool operator==(const Date& a, const Date& b) // equality: nonmember function
{
  return a.day()== b.day() && a.month()== b.month() && a.year()==b.year();
}
```

- Other comparison operators

```
bool operator!=(Date, Date); // inequality
bool operator<(Date, Date); // less than
bool operator>(Date, Date); // greater than
// ...
Date& operator++(Date& d); // increase Date by one day
Date& operator--(Date& d); // decrease Date by one day
Date& operator+=(Date& d, int n); // add n days
Date& operator=(Date& d, int n); // subtract n days
Date operator+(Date d, int n); // add n days
Date operator(Date d, int n); // subtract n days
```

Members and non-members operators

- I prefer to minimize the number of functions that directly manipulate the representation of an object.
  - Bjarne Stroustrup

1

```
// Complex.h
class Complex {
  double re, im;
public:
  Complex& operator+=(Complex a); // needs access to representation
  // ...
};
Complex operator+(const Complex&, const Complex&);
```

2

```
// Complex.cpp
Complex& Complex::operator+=(const Complex& c)
{
  re += c.re;
  im += c.im;
  return *this;
}

Complex operator+(const Complex& c1, const Complex& c2)
{
  Complex r = c1;
  r += c2; // access representation through +=
  return r;
}
```

3

```
void f(Complex x, Complex y)
{
  Complex r1 = x + y ;
  Complex r2 = x;
  r2 += y; // r2.operator+=(y)
  r2 +=z; // r2.operator+=(z)
}
```

accessing vector elements: The Subscription operator

- The get/set member functions: verbose and ugly

1

```
class Vector {
    int sz;
    int* elem;
public:
    int get(int i) const { return elem_[i]; }
    void set(int i, int val) { elem_[i] = val; }
};
```

- Subscription operator:

2

```
class Vector {
    int sz;
    int* elem;
public:
    int& operator[](int i)
    {
        return elem[i];
    }
    int operator[](int i) const
    {
        return elem[i];
    }
};
```

for non-const objects, returns integer variable

for const objects, returns integer value

3

```
void ff(const Vector& cv, Vector& v)
{
    int d = cv[1]; // fine: uses the const []
    cv[1] = 2; // error: uses the const []
    int d = v[1]; // fine: uses the non-const[]
    v[1] = 2; // fine: uses the non-const []
}
```

آلفا ALPHA

Improper use of operator overloading

Improper use of operator overloading

- Operator overloading is like “*syntactic sugar*”.
- Operator overloading helps to program in the language of the *problem domain* rather than in the *language of the machine*.

Improper use of operator overloading

- Operator overloading is like "*syntactic sugar*".
- Operator overloading helps to program in the language of the *problem domain* rather than in the *language of the machine*.

```
class Stack { // stack of character
public:
  void Push(char);
  char Pop();
  bool IsFull() const;
  bool IsEmpty() const;
  // …
};
```

GOOD

Improper use of operator overloading

- Operator overloading is like "*syntactic sugar*".
- Operator overloading helps to program in the language of the *problem domain* rather than in the *language of the machine*.

```
class Stack { // stack of character
public:
  void Push(char);
  char Pop();
  bool IsFull() const;
  bool IsEmpty() const;
  // …
};
```

GOOD

```
class Stack { // stack of character
public:
  void operator+(char);
  char operator-();
  bool IsFull() const;
  bool IsEmpty() const;
  // …
};
```

BAD!

Improper use of operator overloading

- Operator overloading is like "*syntactic sugar*".
- Operator overloading helps to program in the language of the *problem domain* rather than in the *language of the machine*.

```
class Stack { // stack of character
public:
  void Push(char);
  char Pop();
  bool IsFull() const;
  bool IsEmpty() const;
  // …
};
```

GOOD

```
class Stack { // stack of character
public:
  void operator+(char);
  char operator-();
  bool IsFull() const;
  bool IsEmpty() const;
  // …
};
```

BAD!

☞

- When in doubt, choose the non-operator interface.

Thanks for your patience …

A man who asks a question is a fool for minute,
The man who does not ask, is a fool for a life.
- Confucius

Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer.
- Bjarne Stroustrup *programming Principles and Practice Using C++, page 4.*

There is no stupid question, but there is stupid answer.
- Howard Hinnant