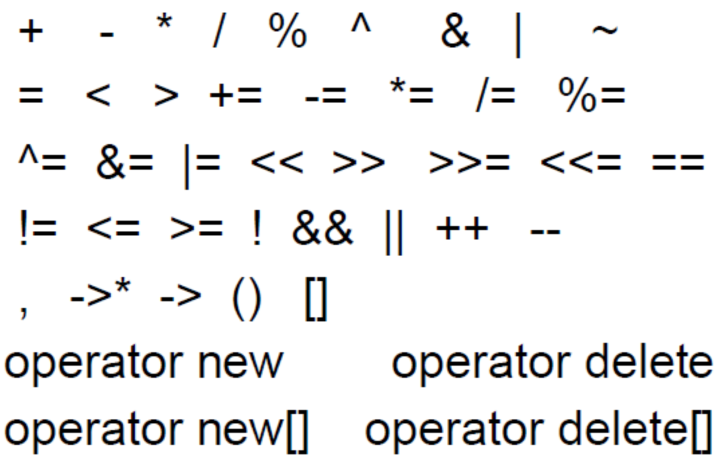
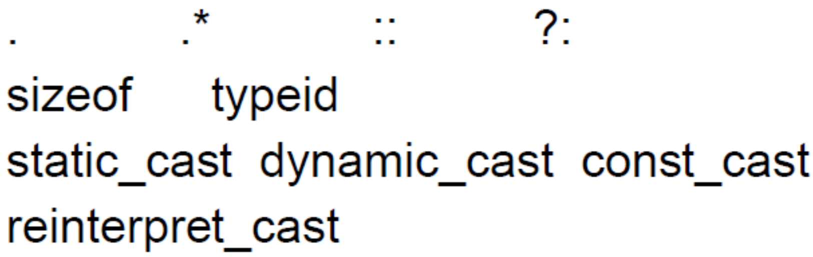
Overload Operator

1 Operators that can be overloaded



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

2 Operators that can not be overloaded



```
.        .*        ::        ?:
sizeof     typeid
static_cast  dynamic_cast  const_cast
reinterpret_cast
```

3 Restriction

- Only existing operators can be overloaded, `eg.` you can not create a `**` operator for exponentiation —— 只能重载已经存在的运算符，而不能自己定义；
- Operators must be overloaded on a **class** or **enumeration type** —— 只能在自己定义的类型中进行运算符重载.
- Overloaded operators must:
  - preserve number of operands;
  - preserve precedence.

4 Overload function

- Just a **function** with an **operator name**!
  - Use the `operator` keyword as a prefix to name

    ```
    operator*(...)
    ```

- Can be **a member function**:
  - **Implicit first argument**, 其实是有两个参数的，因为里成员函数都有一个 `this->`

```
const String String::operator+( const String& that );
```

- Can be a **global (free) function**:
    - Both argument explicit —— 所有的参数都要显式地声明出来

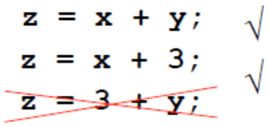
```
const String operator+( const String& r, const String& l );
```

4.1 As Member Functions

```
class Integer {
public:
    Integer( int n = 0 ) : i(n) {}
    const Integer operator+( const Integer& n ) const{
        return Integer(i + n.i);
    }
    ...
private:
    int i;
};
```

```
Integer x(1), y(5), z;
x + y;                  // ====> x.operator+(y);
```

- Implicit first argument;
- Developer must have access to class;
- Members have **full access to all data** in class.
- **No type conversion performed on receiver(运算符左边的对象)**:



```
z = x + y;   √
z = x + 3;   √
~~z = 3 + y;~~
```

- 第三个不对是因为：`+` 左边的这个 `3` 是 receiver，而 `3` 是 `int` 类型，在成员函数重载的时候不会尝试将 `3` 转换成右边 `y` 的 `Integer` 类型；
- 第二个对是因为：`+` 左边的 receiver `x` 是 `Integer` 类型的，于是会在运算的时候将右边的 `3` 先转变成 `Integer` 对象，然后调用类成员于运算符重载函数；

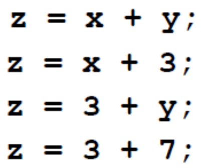
- For **binary operators** `(+, -, *, etc)` member functions require one argument.
- For **unary operators** `(unary -, !, etc)` member functions require no arguments:

```
const Integer operator-() const {
    return Integer(-i);
}
...
z = -x;     // z.operator=(x.operator-());
```

4.2 As Global Functions

```
class Integer {
    friend const Integer operator+( const Integer& lhs, const Integer& rhs );
    ...
}
const Integer operator+( const Integer& lhs, const Integer& rhs ) {
    return Integer( lhs.i + rhs.i );
}
```

- Explicit first function;
  - **binary operators** require two arguments, **unary operators** require one argument
- **Type conversions** performed on **both arguments**;



```
z = x + y;
z = x + 3;
z = 3 + y;
z = 3 + 7;
```

  - 第三个是可以的：编译器会首先检查 `+` 左边的对象是不是 `Integer` 类型的，当发现不是的话，如果是 member function 就报错了，但是是 **global function** 的话就会**尝试将左边转换成右边对象的类型**；
  - 第四个执行的操作：先执行 `int` 类型的 `3+7`，然后用 `10` 构造一个新的 `Integer`，当**有多种可能性的时候编译器会选择做代价最小的**；
- Developer does not need special access to classes, may be made a **friend**.
  - If you don't have access to private data members, then the global function must use the public interface (e.g. accessors).

4.3 Tips: Members vs. Free Functions

- **Unary operators should** be **members**;
- `= () [] -> ->*` **must** be **members**;
- **assignment** operators **should** be **members**;
- All **other binary operators** as **non-members**;

5 Argument Passing & Return Values

5.1 Argument Passing

- If it is **read-only** pass it in **as a** `const` **reference**(except built-ins);
- Make member functions `const` that **don't change the class**(**boolean operators,** `+`**,** `-`, etc);
- For **global functions**, if the **left-hand side changes** pass as a **reference**(assignment operators)
  - `cout << i;` 把 `i` 写到 `cout` 中会修改 `cout` 的状态；

5.2 Return Values

- Select the return type depending on the expected meaning of the operator. For example:
  - For **operator**`+` you need to generate a new object. Return as **a** `const` **object** so the result **cannot be modified as an left value** —— 我们加减法的结果是不能做左值的，所以要返回 `const` 类型；

    ```
    a + b = 6;          // error, cannot happen
    a.operator+(b) = 6; // we should guarantee a.operator+(b) is const so this                             cannot happen(the result cannot be a left value)
    ```

  - **Logical operators** should return `bool` (or `int` for older compilers).

5.3 The prototypes of operators

- `+, -, *, /, %, ^, &, |, ~`：传入什么类型返回就是该类型的 `const` 对象

  ```
  const T operatorX( const T& l, const T& r );
  ```

  - 返回的**不是 reference** 是因为这些操作返回的一定是一个**新的值**；
- `!, &,&, ||, <, <=, ==, >=, >`：逻辑运算返回就是布尔量

  ```
  bool operatorX( const T& l, const T& r );
  ```

- `[]`：取下标操作

  ```
  E& T::operator[]( int index );
  ```

  - 注意返回的时候是个 reference，所有需要将返回对象作为左值的都需要是 reference；
  - 且没有 `const`，因为会做 `a[i] = 6`(**返回的对象要做左值**)；
- Operator `++` and `--`:
  - postfix forms take an `int` argument —— complier will pass in 0 as that `int`:

    ```
    class Integer {
    public:
        ...
        const Integer& operator++();    //prefix++
        const Integer operator++(int);  //postfix++
        const Integer& operator--();    //prefix--
        const Integer operator--(int);  //postfix--
        ...
    };
    ```

```
const Integer& Integer::operator++() {      // ++a
    *this += 1;     // increment
    return *this;   // fetch
}
// int argument not used so leave unnamed so
// won't get compiler warnings
const Integer Integer::operator++( int ) {  // a++
    Integer old( *this );   // fetch, copy constructor
    ++(*this);              // increment, 调用上面的函数做(减少日后可能代码修改)
    return old;             // return
}
```

- using the overloaded `++` and `--`:

```
// decrement operators similar to increment
Integer x(5);
++x;    // calls x.operator++();
x++;    // calls x.operator++(0);
--x;    // calls x.operator--();
x--;    // calls x.operator--(0);
```

  - **User-defined prefix** is **more efficient** than postfix.

- Relational operators:
  - implement `!=` in terms of `==`; implement `>, >=, <=` in terms of `<` —— 只需要定义好 `==` 和 `<`，就别的四个就可以通过调用 `==` 和 `<` 得到;

```
class Integer {
public:
    ...
    bool operator==( const Integer& rhs ) const;
    bool operator!=( const Integer& rhs ) const;
    bool operator<( const Integer& rhs ) const;
    bool operator>( const Integer& rhs ) const;
    bool operator<=( const Integer& rhs ) const;
    bool operator>=( const Integer& rhs ) const;
}
```

-
  ```
  bool Integer::operator==( const Integer& rhs ) const {
      return i == rhs.i;
  }
  // implement lhs != rhs in terms of !(lhs == rhs)
  bool Integer::operator!=( const Integer& rhs ) const {
      return !(*this == rhs);
  }
  bool Integer::operator<( const Integer& rhs ) const {
      return i < rhs.i;
  }
  // implement lhs > rhs in terms of lhs < rhs
  bool Integer::operator>( const Integer& rhs ) const {
      return rhs < *this;
  }
  // implement lhs <= rhs in terms of !(rhs < lhs)
  bool Integer::operator<=( const Integer& rhs ) const {
  ```

```cpp
    return !(rhs < *this);
}
// implement lhs >= rhs in terms of !(lhs < rhs)
bool Integer::operator>=( const Integer& rhs ) const {
    return !(*this < rhs);
}
```

- A stream extractor —— **是固定的格式**:
  - **Has to** be a **2-argument free** function: first argument is **an** `istream&`, second argument is **a** `reference` **to a value(not** `const`**)** —— 这是一个双目运算符 `istream && object`:

    ```cpp
    istream& operator>>( istream& is, T& obj ) {
        // specific code to read obj
        return is;
    }
    ```

  - **Must** return **an** `istream&` for chaining:

    ```cpp
    cin >> a >> b >> c;
    ((cin >> a) >> b) >> c;
    ```

- A stream inserter —— **是固定的格式**:
  - **Has to** be a **2-argument free** function: first argument is **an** `ostream&`, second argument is **a** `reference` **to a value(**`const`**)** —— 这是一个双目运算符 `ostream && object`:

    ```cpp
    ostream& operator<<( ostream& os, const T& obj ) {
        // specific code to write obj
        return os;
    }
    ```

  - **Must** return **an** `istream&` for chaining:

    ```cpp
    cout << a << b << c;
    ((cout << a) << b) << c;
    ```

- Creating **manipulators**:

```cpp
// skeleton for an output stream manipulator
ostream& manip( ostream& out ) {
    ...
    return out;
}
ostream& tab ( ostream& out ) {
    return out << '\t';
}
cout << "Hello" << tab << "World!" << endl;
```

5.4 Assignment Operator `=`

> Automatic operator= creation

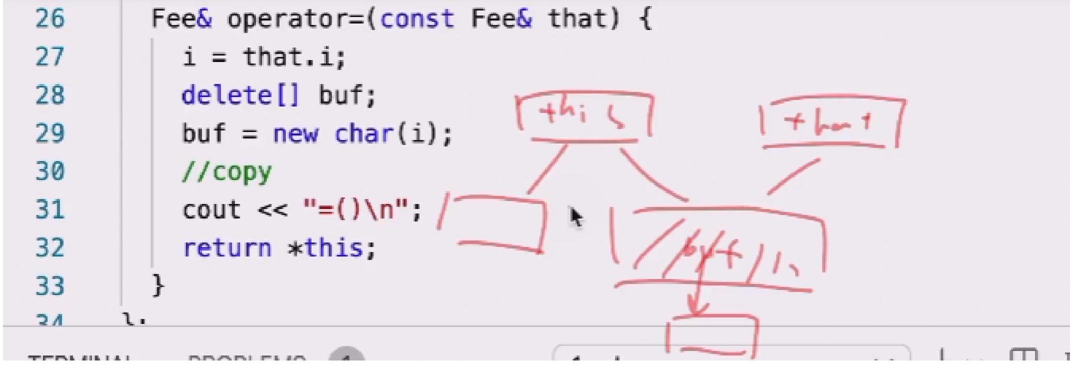
- The compiler will **automatically create** a `type::operator=(type)` if you don't make one;
  - member-wise assignment.

> Assignment Operator Overloading

- **Must** be a **member function**;
- **Return** a **reference** to `*this` —— **模板是固定的**:

```
T& T::operator=( const T& rhs ) {
    // check for self assignment!!!!!!!!
    if ( this != &rhs ) {
        // perform assignment
    }
    return *this;
}
//This checks address vs. check value (*this != rhs)
```

  - 如果类不涉及指针的话，直接让编译器产生默认的 member-wise 的赋值函数就可以；
  - 如果类中**涉及指针**的话，就必须要**自己写**一个 `T& T::operator=( const T& rhs )` 重载函数，并且在这个重载函数中**首先就要判断传进来的** `rhs` **和** `*this` **相不相等**，因为如果相等的话会出现：



```
    Fee& operator=(const Fee& that) {
        i = that.i;
        delete[] buf;
        buf = new char(i);
        //copy
        cout << "=()\n";
        return *this;
    }
};
```

删掉 `this` 指向的东西的同时将 `rhs` 指向的东西删掉了，导致后面就没有东西来进行赋值了；

5.5 Value Classes

- Appear to be primitive data types, can be passed to and returned from functions;
- Have overloaded operators(often);
- Can be converted to and from other types.

---

6 User-defined Type Conversions

- A conversion operator can be used to convert an object of one class into an object of another class or a build-in type;
- Compilers perform implicit conversions using:
  - Single-argument constructors, `eg. PathNmae xyz(abc);`;

    - Implicit type conversion operators, `eg. xyz = abc;`.

      ```
      class PathName {
          string name;
      public:
          // or could be multi-argument with defaults
          PathName( const string& );  // Single-argument constructors
          ~ PathName();
      };
      ...
      string abc("abc");
      PathName xyz(abc);  // OK!
      xyz = abc;          // OK abc => PathName, Implicit type conversion
      ```

- We can use **keyword** `explicit` to prevent implicit conversions:

  ```
  class PathName {
      string name;
  public:
      explicit PathName( const string& );
      ~ PathName();
  };
  ...
  string abc("abc");
  PathName xyz(abc);  // OK!
  xyz = abc;          // error!
  ```

6.1 Conversion Operations

- Function will be **called automatically**, and **return type** is the **same as function name** —— 不需要显示地指出返回类型(和构造函数一样)，默认返回类型和函数名称一样：

  ```
  class Rational {
  public:
      ...
      operator double() const; // Rational to double
  }
  Rational::operator double() const {
      return numerator_/(double)denominator_;
  }
  Rational r(1,3); double d = 1.3 * r; // r=>double
  ```

    - General form of conversion ops:

      ```
      X::operator T();
      ```

        - operator name is any type descriptor;
        - no explicit arguments, and no return type;
        - compiler will use it as a type conversion from `X ==> T`.

In general, no!!! —— 一般不要去使用！！！

☑ Because lots of problems when functions are called unexpectedly;

- ☑ Use explicit conversion functions. `eg.` in class `Rational` instead of the conversion operator, declare a member function:

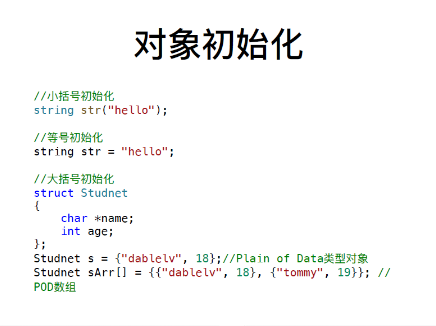
  ```
  double toDouble() const;
  ```

6.2 Overloading and type conversion

- C++ checks each argument for a "best match":
  1. Exact match is cost-free —— 完全匹配消耗最少；
  2. Matches involving built-in conversions —— 和 build-in 转换匹配上了；
  3. User-defined type conversion —— 用户自己定义的转换是消耗最大的，因为要调用函数；

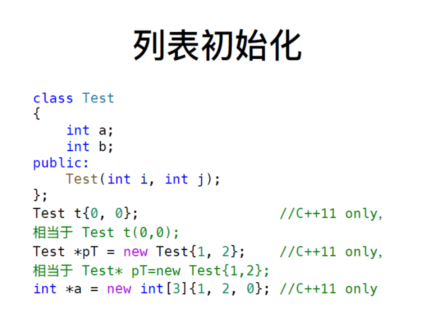
---

7 Ways of Initialization



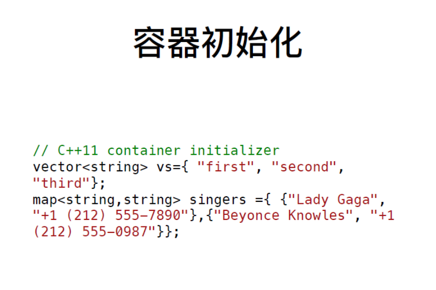
对象初始化

```
//小括号初始化
string str("hello");

//等号初始化
string str = "hello";

//大括号初始化
struct Studnet
{
    char *name;
    int age;
};
Studnet s = {"dablelv", 18};//Plain of Data类型对象
Studnet sArr[] = {{"dablelv", 18}, {"tommy", 19}}; //
POD数组
```



列表初始化

```
class Test
{
    int a;
    int b;
public:
    Test(int i, int j);
};
Test t{0, 0};                    //C++11 only,
相当于 Test t(0,0);
Test *pT = new Test{1, 2};    //C++11 only,
相当于 Test* pT=new Test{1,2};
int *a = new int[3]{1, 2, 0}; //C++11 only
```



容器初始化

```
// C++11 container initializer
vector<string> vs={ "first", "second",
"third"};
map<string,string> singers ={ {"Lady Gaga",
"+1 (212) 555-7890"},{"Beyonce Knowles", "+1
(212) 555-0987"}};
```

---

8 type of function parameters and return value

- Pass in an object if you want to store it;
- Pass in a const pointer or reference if you want to get the values;
- Pass in a pointer or reference if you want to do something to it;
- Pass out an object if you create it in the function;
- Pass out pointer or reference of the passed in only;
- Never new something and return the pointer.

---

9 Left Value vs Right Value

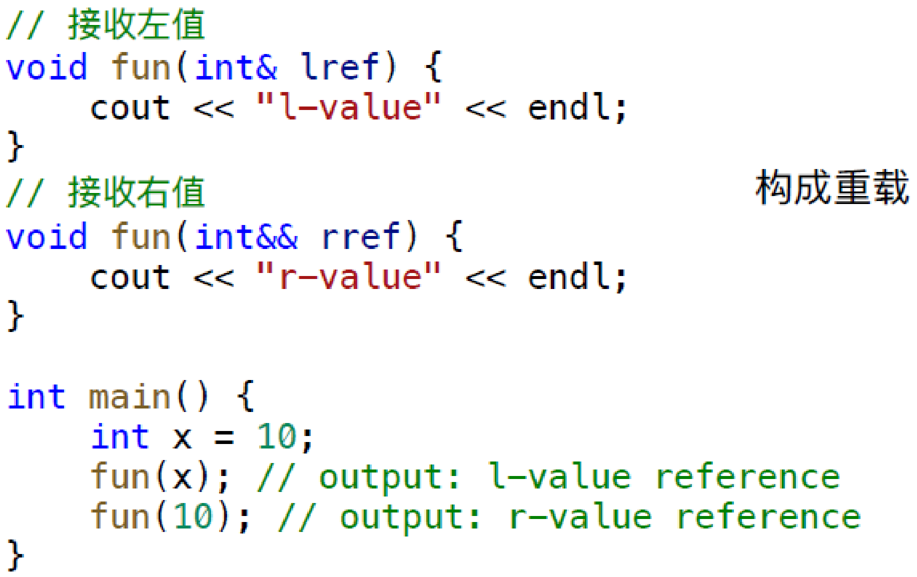
- 可以简单地认为能出现**在赋值号左边的**都是左值：
  - 变量本身、引用、`*`、`[]` 运算的结果；

- 只能出现**在赋值号右边的**都是右值：
  - 字面量、表达式；
- **引用只能接受左值** → 引用是左值的别名；
- 调用函数时的传参相当于参数变量在调用时的初始化；

9.1 右值引用

- `int x = 20;`：左值
- `int&& rx = x*2;`：`x*2` 的结果是一个右值，`rx` 延长其生命周期；
- `int y = rx + 2;`：因此我们可以重用它；
- `rx = 100;`：一旦你初始化一个右值引用变量，该变量就成为了一个左值，可以被赋值；
- `int&& rrx1 = x;`：**非法的，右值引用无法被左值初始化**；
  - `const int&& rrx2 = x;`：**非法，右值引用无法被左值初始化**；

9.2 右值参数



```
// 接收左值
void fun(int& lref) {
    cout << "l-value" << endl;
}
// 接收右值
void fun(int&& rref) {
    cout << "r-value" << endl;
}

int main() {
    int x = 10;
    fun(x); // output: l-value reference
    fun(10); // output: r-value reference
}
```

- 所以其实右值引用的作用就在于在函数调用的时候降低开销；