Class Analysis and Design

# 4 Class Analysis & Design



195

## Overview of this section

- Contents of this chapter
  - **Object Oriented Analysis and Design** – A first shot at decomposing your problem into classes
  - **Designing the class interface** – Style guide and common issues
  - **Operator overloading** – Making your class behave more like built-in types
  - **Friends** – Breaking access patterns to enhance encapsulation



196

## Class Analysis and Design

- We now understand the basics of writing classes
  - Now it's time to think about how to decompose your problem into classes
- Writing good OO software involves 3 separate steps
  1. **Analysis**
  2. **Design**
  3. **Programming**
  - You can do them formally or informally, well or poorly, but you can't avoid them
- Analysis
  - How to divide up your problem in classes
  - What should be the functionality of each class
- Design
  - What should the interface of your class look like?

197

## Analysis – Find the class

- OO Analysis subject of many text books, many different approaches
  - Here some basic guidelines

1. Try to describe briefly in plain English (or Dutch) what you intend your software to do
   - Rationale – This naturally makes you think about your software in a high abstraction level

2. Associate software objects with natural objects ('objects in the application domain')
   - Actions translate to member functions
   - Attributes translate to data members

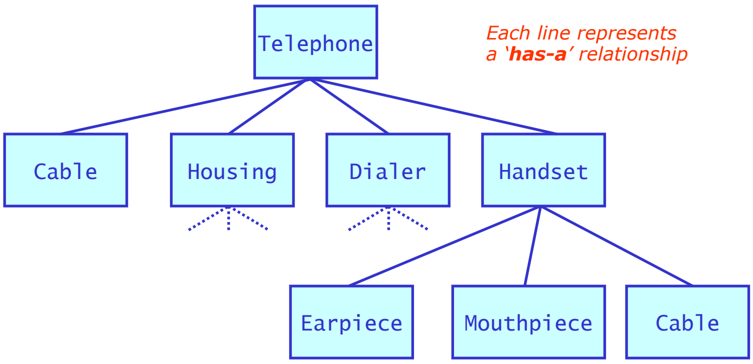
3. Make hierarchical ranking of objects using 'has-a' relationships
   - Example: a '`BankAccount`' has-a '`Client`'
   - Has-a relationships translate into data members that are objects

4. Iterate! Nobody gets it right the first time

198

## Analysis – A textbook example

- Example of telephone hardware represented as class hierarchy using 'has-a' relationships
  - Programs describing or simulating hardware usually have an intuitive decomposition and hierarchy

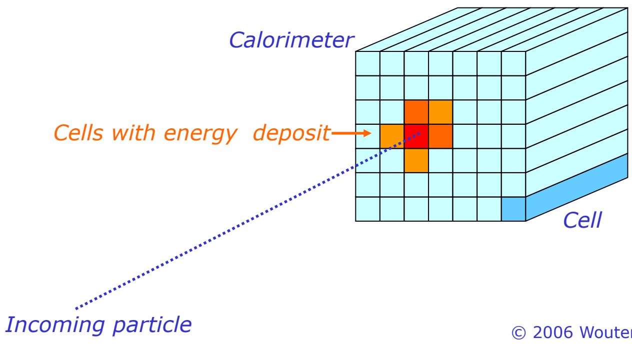





199

## Analysis – Example from High Energy Physics

- Real life often not so clean cut
- Example problem from High Energy physics
  - We have a file with experimental data from a calorimeter.
  - A calorimeter is a HEP detector that detects energy through absorption. A calorimeter consists of a grid of detector modules (cells) that each individually measure deposited energy

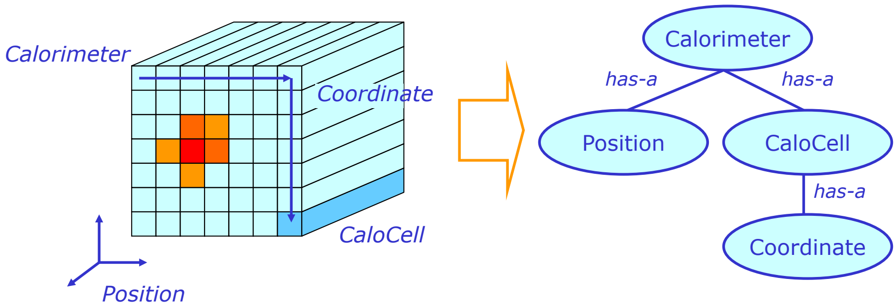





200

## Analysis – Example from High Energy Physics

- First attempt to identify objects in data processing model and their containment hierarchy
  - Calorimeter global position and cell coordinates are not physical objects but separate logical entities so we make separate classes for those too

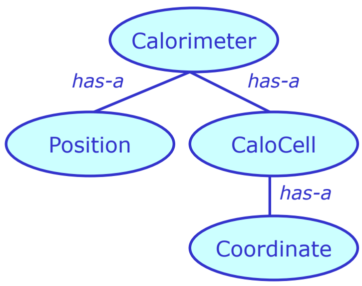





201

## Analysis – Example from High Energy Physics

- Key Analysis sanity check – Can we describe what each object ***is***, in addition to what it does?
  - Answer: yes

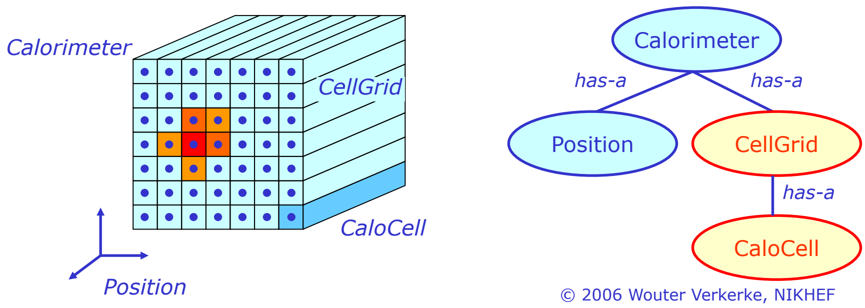





202

## Analysis – Example from High Energy Physics

- Iterating the design – are there other/better solutions?
  - Remember 'strong cohesion' and 'loose coupling'
  - Try different class decomposition, moving functionality from one class to another
- Example of alternative solution
  - We can store the `CaloCells` in an intelligent container class `CellGrid` that mimics a 2D array and keeps track of coordinates

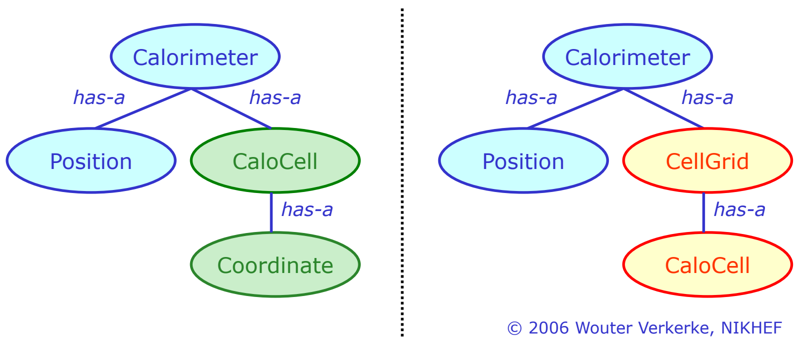





203

## Analysis – Example from High Energy Physics

- Which solution is better?
  - Source of ambiguity: cell coordinate not really intrinsic property of calorimeter cell
  - Path to solution: what are cell coordinates used for? Import for insight in best solution. Real-life answer: to find adjacent (surrounding cells)
  - Solution: Adjacency algorithms really couple strongly to layout of cells, not to property of individual cells → design with layout in separate class probably better

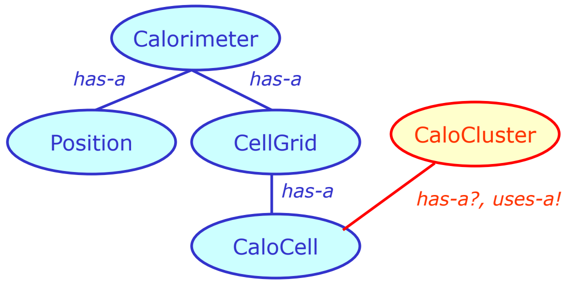





204

## Extending the example – Has-A vs Uses-A

- Next step in analysis of calorimeter data is to reconstruct properties of incoming particles
  - Reconstruct blobs of energy deposited into multiple cells
  - Output stored in new class `CaloCluster`, which stores properties of cluster and refers back to cells that form the cluster




  - Now we run into some problems with 'has-a' semantics: All `CaloCells` in `Calorimeter` are owned by `Calorimeter`, so `CaloCluster` doesn't really 'have' them. Solution: '**Uses-A**' semantic.
  - A '**Uses-A**' relation translates into a pointer or reference to an object

205

## Summary on OO analysis

- Choosing classes: You should be able to say what a class **is**
  - A 'Has-A' relation translates into data members, a 'Uses-A' relation into a pointer
  - Functionality of your natural objects translates in member functions
- Be wary of complexity
  - Signs of complexity: repeated identical code, too many function arguments, too many member functions, functions with functionality that cannot be succinctly described
  - A complex class is difficult to maintain → Redesign into smaller units
- There may not be a unique or 'single best' decomposition of your class analysis
  - Such is life. Iterate your design, adapt to new developments
- We'll revisit OOAD again in a while when we will discuss polymorphism and inheritance which open up many new possibility (and pitfalls)



206

## The art of proper class design

- Class Analysis tells you what functionality your class should have
- Class Design now focuses on how to package that best
- Focus: Make classes easy to use
  - Robust design: copying objects, assigning them (even to themselves) should not lead to corruption, memory leaks etc
  - Aim for intuitive behavior: mimic interface of built-in types where possible
  - Proper functionality for 'const objects'
- Reward: better reusability of code, easier maintenance, shorter documentation
- And remember: Write the interface first, then the implementation
  - While writing the interface you might still find flaws or room for improvements in the design. It is less effort to iterate if there is no implementation to data



207

## The art of proper class design

- Focus on following issues next
  - **Boilerplate class design**
  - **Accessors & Modifiers** – Proper interface for const objects
  - **Operator overloading**
  - **Assignment** – Why you need it
  - Overloading **arithmetic, and subscript operators**
  - Overloading **conversion operators**, use of explicit
  - Spilling your guts – **friends**



208

## Check list for class interface

- A boilerplate class design
- When writing a class it helps to group member functions into the following categories
  - **Initialization** – Constructors and helper functions
  - **Assignment**
  - **Cleanup** – Destructors and helper functions
  - **Accessors** – Function providing read-only access to data members
  - **Modifiers** – Functions that allow to modify data members
  - **Algorithmic functions**
  - **I/O functions**
  - **Error processing functions**



209

## Accessor / modifier pattern

- For each data member that is made publicly available implement an accessor and a modifier
- Pattern 1 – Encapsulate read & write access in separate functions
  - Complete control over input and output. Modifier can be `protected` for better access control and modifier can validate input before accepting it
  - Note that returning large data types by value is inefficient. Consider to return a const reference instead

```
class Demo {
private:
   float _val ;
public:
   // accessor
   float getVal() const {
      return _val ;
   }
   // modifier
   void setVal(float newVal) {
     // Optional validity checking goes here
      _val = newVal ;
   }
} ;
```

const here is important otherwise this will fail

```
const Demo demo ;
demo.getVal() ;
```



210

## Accessor / modifier pattern

- Pattern 2 – Return reference to internal data member
  - Must implement both const reference and regular reference!
  - Note that no validation is possible on assignment. Best for built-in types with no range restrictions or data members that are classes themselves with built-in error checking and validation in their modifier function

```
class Demo {
private:
   float _val ;

public:
   float& val() { return _val ; }
   const float& val() const { return _val ; }

} ;
```

non-const version can be used on LHS of equal sign

```
Demo demo ;
demo.val() = 5.3 ;
```

const version here is essential, otherwise code below will fail

```
const Demo demo ;
float demoVal = demo.val() ;
```

211

## Making classes behave like built-in objects

- Suppose we have written a 'class complex' that represents complex numbers
  - Execution of familiar math through add(),multiply() etc member functions easily obfuscates user code

```
complex a(3,4), b(5,1) ;

b.multiply(complex(0,1)) ;
a.add(b) ;
a.multiply(b) ;
b.subtract(a) ;
```

  - Want to redefine meaning of C++ operators +,* etc to perform familiar function on newly defined classes, i.e. we want compiler to automatically translate:

```
c = a * b ;   ➡   c.assign(a.multiply(b)) ;
```

- Solution: C++ operator overloading



212

## Operator overloading

- In C++ **operations are functions too**, i.e.

*What you write*

```
complex c = a + b;
```

*What the compiler does*

```
c.operator=(operator+(a,b));
```

- Operators can be both regular functions as well as class member functions
  - In example above `operator=()` is implemented as member function of class complex, `operator+()` is implemented as global function
  - You have free choice here, `operator+()` can also be implemented as member function in which case the code would be come

```
c.operator=(a.operator+(b));
```

  - Design consideration: member functions (including operators) can access 'private' parts, so operators that need this are easier to implement as member functions
    - More on this in a while…



213

## An assignment operator – declaration

- Lets first have a look at implementing the assignment operator for our fictitious class complex
- Declared as member operator of class complex:
  - Allows to modify left-hand side of assignment
  - Gives access to private section of right-hand side of assignment

```
class complex {
public:
   complex(double r, double i) : _r(r), _i(i) {} ;
   complex& operator=(const complex& other) ;

private:
   double _r, _i ;
} ;
```



214

## An assignment operator – implementation

**Copy content of other object**
It is the same class, so you have access to its private members

**Handle self-assignment explicitly**
It happens, really!

```
complex& complex::operator=(const complex& other) {

   // handle self-assignment
   if (&other == this) return *this ;

   // copy content of other
   _r = other._r ;
   _i = other._i ;

   // return reference to self
   return *this ;
}
```

**Return reference to self**
Takes care of chain assignments

215

## An assignment operator – implementation

**Copy content of other object**
It is the same class, so you have access to its private members

**Handle self-assignment explicitly**
It happens, really!

```
complex& complex::operator=(const complex& other) {

   // handle self-assignment
   if (&other == this) return *this ;
```

**Why ignoring self-assignment can be bad**
Image you store information in a dynamically allocated array that needs to be reallocated on assignment…

```
A& A::operator=(const A& other) {
   delete _array ;
   _len = other._len;
   _array = new int[other._len] ;
   // Refill array here
   return *this ;
}
```

Oops if (other==*this)
**you just deleted your own array!**

216

## An assignment operator – implementation

**Why you should return a reference to yourself**
Returning a reference to yourself allows chain assignment

```
complex a,b,c ;
a = b = c ;
```

→

```
complex a,b,c ;
a.operator=(b.operator=(c)) ;
```

Returns reference to b

*Not mandatory, but essential if you want to mimic behavior of built-in types*

```
   // handle self-assignment
   if (&other == this) return *this ;

   // copy content of other
   _r = other._r ;
   _i = other._i ;

   // return reference to self
   return *this ;
}
```

**Return reference to self**
Takes care of chain assignments

217

## The default assignment operator

- The assignment operator is like the copy constructor:
  ***it has a default implementation***
  - Default implementation calls assignment operator for each data member
- If you have data member that are pointers to 'owned' objects this will create problems
  - Just like in the copy constructor
- Rule: If your class owns dynamically allocated memory or similar resources you should implement your own assignment operator
- You can disallow objects being assigned by declaring their assignment operator as 'private'
  - Use for classes that should not copied because they own non-assignable resources or have a unique role (e.g. an object representing a file)



218

## Example of assignment operator for owned data members

```
class A {
private:
   float* _arr ;
   int _len ;
public:
   operator=(const A& other) ;
} ;
```

**C++ default operator=()**

```
A& operator=(const A& other) {
  if (&other==this) return *this;
  _arr = other._arr ;
  _len = other._len ;
  return *this ;
}
```

**YOU DIE.**

If other is deleted before us, _arr will point to garbage. Any subsequent use of self has undefined results

If we are deleted before other, we will delete _arr=other._arr, which is not owned by us: other._arr will point to garbage and will attempt to delete array again

**Custom operator=()**

```
A& operator=(const A& other) {
  if (&other==this) return *this;
  _len = other._len ;
  delete[] _arr ;
  _arr = new int[_len] ;
  int i ;
  for (i=0; i<len ; i++) {
     _arr[i] = other._arr[i] ;
  }
  return *this ;
}
```



219

## Overloading other operators

- Overloading of operator=() mandatory if object owns other objects
- Overloading of other operators voluntary
  - Can simplify use of your classes (example: class complex)
  - But don't go overboard – Implementation should be congruent with meaning of operator symbol
    - E.g. don't redefine operator^() to implement exponentiation
  - Comparison operators (<,>,==,!=) useful to be able to put class in sortable container
  - Addition/subtraction operator useful in many contexts: math objects, container class (add new content/ remove content)
  - Subscript operator[] potentially useful in container classes
  - Streaming operators <<() and operator>>() useful for printing in many objects
- Next: Case study of operator overloading with a custom string class



220

## The custom string class

- Example string class for illustration of operator overloading

```
class String {
private:
  char* _s ;
  int _len ;

  void insert(const char* str) { // private helper function
     _len = strlen(str) ;
     if (_s) delete[] _s ;
     _s = new char[_len+1] ;
    strcpy(_s,str) ;
  }

public:
  String(const char* str= "") : _s(0) { insert(str) ; }
  String(const String& a)  : _s(0) { insert(a._s) ; }
  ~String() { if (_s) delete[] _s ; }

  int length() const { return _len ; }
  const char* data() const { return _s ; }
  String& operator=(const String& a) {
    if (this != &a) insert(a._s) ;
    return *this ;
   }
} ;
```

**Data members, array & length**



221

## The custom string class

- Example string class for illustration of operator overloading

```
class String {
private:
  char* _s ;
  int _len ;

  void insert(const char* str) { // private helper function
     _len = strlen(str) ;
     if (_s) delete[] _s ;
     _s = new char[_len+1] ;
    strcpy(_s,str) ;
  }

public:
  String(const char* str= "") : _s(0) { insert(str) ; }
  String(const String& a)  : _s(0) { insert(a._s) ; }
  ~String() { if (_s) delete[] _s ; }

  int length() const { return _len ; }
  const char* data() const { return _s ; }
  String& operator=(const String& a) {
    if (this != &a) insert(a._s) ;
    return *this ;
   }
} ;
```

**Delete old buffer, allocate new buffer, copy argument into new buffer**



222

## The custom string class

- Example string class for illustration of operator overloading

```
class String {
private:
  char* _s ;
  int _len ;

  void insert(const char* str) { // private helper function
     _len = strlen(str) ;
     if (_s) delete[] _s ;
     _s = new char[_len+1] ;
    strcpy(_s,str) ;
  }

public:
  String(const char* str= "") : _s(0) { insert(str) ; }
  String(const String& a)  : _s(0) { insert(a._s) ; }
  ~String() { if (_s) delete[] _s ; }

  int length() const { return _len ; }
  const char* data() const { return _s ; }
  String& operator=(const String& a) {
    if (this != &a) insert(a._s) ;
    return *this ;
   }
} ;
```

**Ctor Dtor**



223

## The custom string class

- Example string class for illustration of operator overloading

```
class String {
private:
  char* _s ;
  int _len ;

  void insert(const char* str) { // private helper function
     _len = strlen(str) ;
     if (_s) delete[] _s ;
     _s = new char[_len+1] ;
    strcpy(_s,str) ;
  }

public:
  String(const char* str= "") : _s(0) { insert(str) ; }
  String(const String& a)  : _s(0) { insert(a._s) ; }
  ~String() { if (_s) delete[] _s ; }

  int length() const { return _len ; }
  const char* data() const { return _s ; }
  String& operator=(const String& a) {
    if (this != &a) insert(a._s) ;
    return *this ;
   }
} ;
```

**Overloaded assignment operator**

224

## Overloading operator+(), operator+=()

- Strings have a natural equivalent of addition
  - "A" + "B" = "AB"
  - Makes sense to implement `operator+`
- Coding guideline: if you implement +, also implement +=
  - In C++ they are separate operators.
  - Implementing + will not automatically make += work.
  - Implementing both fulfills aim to mimic behavior of built-in types
- Practical tip: Do `operator+=()` first.
  - It is easier
  - `Operator+` can trivially be implemented in terms of `operator+=` (code reuse)



225

## Overloading operator+(), operator+=()

- Example implementation for String
  - Argument is `const` (it is not modified after all)
  - Return is reference to self, which allows chain assignment

```
class String {
public:
  String& operator+=(const String& other) {
    int newlen = _len + other._len ;     // calc new length
    char* newstr = new char[newlen+1] ; // alloc new buffer

    strcpy(newstr,_s) ;                 // copy own contents
    strcpy(newstr+_len,other._s) ;      // append new contents

    if (_s) delete[] _s ;               // release orig memory

    _s = newstr ;                       // install new buffer
    _len = newlen ;                     // set new length
    return *this ;
  }
} ;
```



226

## Overloading operator+(), operator+=()

- Now implement `operator+()` using `operator+=()`
  - Operator is a global function rather than a member function – no privileged access is needed to String class content
  - Both arguments are `const` as neither contents is changed
  - Result string is passed by value

```
String operator+(const String& s1, const String& s2) {
    String result(s1) ; // clone s1 using copy ctor
    result += s2 ;      // append s2
    return result ;     // return new result
}
```



227

## Overloading operator+() with different types

- You can also add heterogeneous types with `operator+()`
  - Example: `String("A") + "b"`
- Implementation of heterogeneous operator+ similar
  - Illustration only, we'll see later why we don't need it in this particular case

```
String operator+(const String& s1, const char* s2) {
    String result(s1) ;       // clone s1 using copy ctor
    result += String(s2) ;    // append String converted s2
    return result ;           // return new result
}
```

- NB: Arguments of `operator+()` do not commute

```
operator+(const& A, const& B)!=operator+(const& B, const& A)
```

  - If you need both, implement both



228

## Working with class String

- Demonstration of operator+ use on class String

```
// Create two strings
String s1("alpha") ;
String s2("bet") ;

// Concatenate strings into 3rd string
String s3 = s1+s2 ;

// Print concatenated result
cout << s1+s2 << endl ;
```

Implicit conversion by compiler

```
cout << String(s1+s2) << endl ;
```

- Compare ease of use (*including* correct memory management) to join() functions of exercise 2.1…



229

## Overloading comparison operators ==,!=,<,>

- Comparison operators make sense for strings
  - "A" != "B", "Foo" == "Foo", "ABC" < "XYZ"
  - Comparison operators are essential interface to OO sorting
- Example implementation
  - Standard Library function `strcmp` returns 0 if strings are identical, less than 0 if s1<s2, and greater than 0 if s1>s2
  - Input arguments are `const` again
  - Output type is `bool`
  - Operators <,>,<=,>= similar

```
bool operator==(const String& s1, const String& s2) {
  return (strcmp(s1.data(),s2.data())==0) ;
}

bool operator!=(const String& s1, const String& s2) {
  return (strcmp(s1.data(),s2.data())!=0) ;
}
```



230

## Overloading subscript operators

- Subscript operators make sense for indexed collections such as strings
  - `String("ABCD")[2] = 'C'`
- Example implementation for String
  - Non-const version allows `string[n]` to be use as *lvalue*
  - Const version allows access for const objects

```
char& String::operator[](int i) {
    // Don't forget range check here
    return _s[i] ;
}

const char& String::operator[](int i) const {
    // Don't forget range check here
    return _s[i] ;
}
```



231

## Overloading subscript operators

- Note 1: ***Any*** argument type is allowed in []
  - Example

```
class PhoneBook {
public:
   int PhoneBook::operator[](const char* name) ;
} ;

void example() {
  PhoneBook pbook ;
  pbook["Bjarne Stroustrup"] = 0264524 ;
  int number = pBook["Brian Kernigan"] ;
}
```

  - Powerful tool for indexed container objects
  - More on this later in the Standard Template Library section
- Note 2: C++ does not have multi-dimensional array operator like array[5,3]
  - Instead it has array[5][3] ;
  - If you design a container with multi-dimensional indexing consider overloading the () operator, which works exactly like the [] operator, except that it allows multiple arguments



232

## Overloading conversion operators

- Conversions (such as `int` to `float`) are operators too!
- Sometimes it makes sense to define custom conversions for your class
  - Example: `String` → `const char*`, `const char*` → `String`
- General syntax for conversions from `ClassA` to `ClassB`

```
class ClassA {
   operator ClassB() const ; // conversion creates copy
                             // so operation is const
};
```

- Example implementation for class String

```
String::operator const char*() const {
   return _s ;
}
```



233

## Using conversion operators

- Conversion operators allow the compiler to convert types automatically for you.
  - Example

```
int strlen(const char* str) ; // Standard Library function
String foo("Hello World") ;
int len = strlen(foo) ;
```

```
int strlen(const char* str) ; // Standard Library function
String foo("Hello World") ;
int len = strlen(foo.operator const char*()) ;
```

- Constructors aid the automatic conversion process for reverse conversion from (from another type to yourself)
  - Example: allows automatic conversion from 'const char*' to String

```
class String {
   String(const char* str) ;
};
```

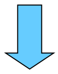


234

## How conversion operators save you work

- Remember that we defined `operator+(const& String, const char*)`
  - It turns out we don't need it if String to 'const char*' conversion is defined
  - Compiler automatically fills in the necessary conversions for you

```
String s("Hello") ;
String s2 = s + " World" ;
```



```
String s("Hello") ;
String s2 = s + String(" World") ;
```

  - ***No need for our operator+(const String&, const char*).***
  - Of course if we can define a dedicated operator that is ***computationally more efficient*** we should still implement it. The compiler will use the dedicated operator instead

235

## Curbing an overly enthusiastic compiler

- Suppose you want define the constructor

```
class String {
   String(const char*) ;
};
```

  but you do not want to compiler to use it for automatic conversions

- Solution: make the constructor `explicit`

```
class String {
   explicit String(const char*) ;
};
```

  - Useful in certain cases



236

## Recap on operator definition

- Operators can be implemented as
  - Global functions
  - Member functions
- For *binary* operators a member function implementation always binds to the *left argument*
  - I.e. 'a + b' → a.operator+(b)
- Rule of thumb:
  - Operators that modify an object should be member functions of that object
  - Operators that don't modify an object can be either a member function or a global function
- But what about operators that modify the rightmost argument?
  - Example cin >> phoneBook → operator>>(cin,phoneBook)



237

## What friends are for

- But what about operators that modify the rightmost argument?
  - Example cin >> phoneBook → operator>>(cin,phoneBook)
  - Sometimes you can use public interface to modify object (e.g. see string example)
  - Sometimes this is not desirable (e.g. interface to reconstitute object from stream is considered private) – what do you do?
- Solution: make friends
  - A friend declaration allows a specified class or function to access the private parts of a class
  - A global function declared as friend does ***NOT*** become a member function; it is only given the same access privileges

```
class String {
   public:
     String(const char*=“”) ;
   private:
     friend istream& operator>>(istream&, String&) ;
} ;
```

238

## Friend and encapsulation

- Worked out string example

```
class String {
   public:
     String(const char*="") ;
   private:
     char* _buf ;
     int _len ;
     friend istream& operator>>(istream&, String&) ;
} ;

istream& operator>>(istream& is, String& s) {
  const int bufmax = 256 ;
  static char buf[256] ;
  is >> buf ;
  delete[] s._buf ;              // Directly
  s._len = strlen(buf) ;         // manipulate
  s._buf = new char[s._len+1] ; // private members
  strcpy(s._buf,buf) ;           // of String s
  return is ;
}
```



239

## Friends and encapsulation

- Friends technically break encapsulation, but when properly used they enhance encapsulation
  - Example: class String and global operator>>(istream&,String&) are really a single module (strong cohesion)
  - Friend allow parts of single logical module to communicate with each other without exposing private interface to the outer world
- Friend declarations are allowed for functions, operators and classes
  - Following declaration makes all member functions of class StringManipulator friend of class String

```
class String {
   public:
     String(const char*="") ;
   private:
     friend class StringManipulator ;
} ;
```

240

## Class string

- The C++ Standard Library provides a `class string` very similar to the example `class String` that we have used in this chapter
  - Nearly complete set of operators defined, internal buffer memory expanded as necessary on the fly
  - Declaration in `<string>`
  - Example

```
string dirname("/usr/include") ;
string filename ;

cout << "Give first name:" ;

// filename buffer will expand as necessary
cin >> filename ;

// Append char arrays and string intuitively
string pathname = dirname + "/" + filename ;

// But conversion string → char* must be done explicitly
ifstream infile(pathname.c_str()) ;
```

241