Introduction to C++ and Object Oriented Programming

Wouter Verkerke (NIKHEF)

v39 – Edition for 2007 UvA APP Master course

0 Introduction & Overview

Intended audience and scope of course

- This course is targeted to students with some programming experience in procedural (i.e. non-OO) programming languages like Fortran, C, Pascal
  - No specific knowledge of C, C++ is assumed

- This course will cover
  - Basic C/C++ syntax, language features
  - Basics of object oriented programming

- This course has some extra focus on the application of C++ in (High Energy) Physics
  - Organized processing and analysis of data
  - Focus mostly in exercises

Programming, design and complexity

- The goal of software – to solve a particular problem
  - E.g. computation of numeric problems, maintaining an organized database of information, finding the Higgs etc..

- Growing computational power in the last decades have allowed us to tackle more and more complex problems

- As a consequence software has also grown more powerful and complex
  - For example Microsoft Windows XP OS, last generation video games, often well over 1.000.000 lines of source code
  - Growth also occurs in physics: e.g. collection of software packages for reconstruction/analysis of the BaBar experiment is ~6.4M lines of C++

- How do we deal with such increasing complexity?

Programming philosophies

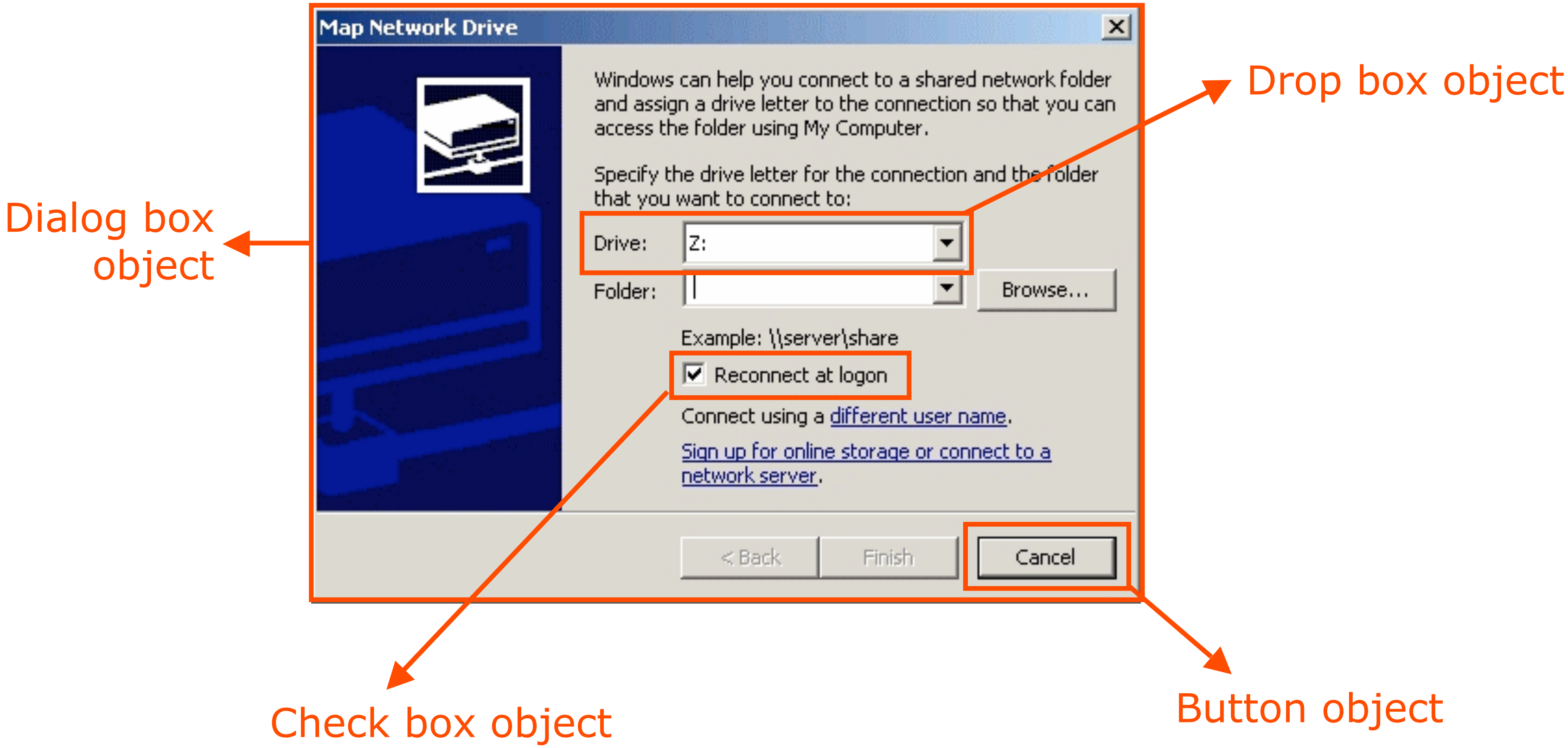
- Key to successfully coding complex systems is break down code into smaller modules and minimize the dependencies between these modules

- Traditional programming languages (C, Fortran, Pascal) achieve this through procedure orientation
  - Modularity and structure of software revolves around 'functions' encapsulate (sub) algorithms
  - Functions are a major tool in software structuring but leave a few major design headaches

- Object-oriented languages (C++, Java,...) take this several steps further
  - Grouping data and associated functions into objects
  - Profound implications for modularity and dependency reduction

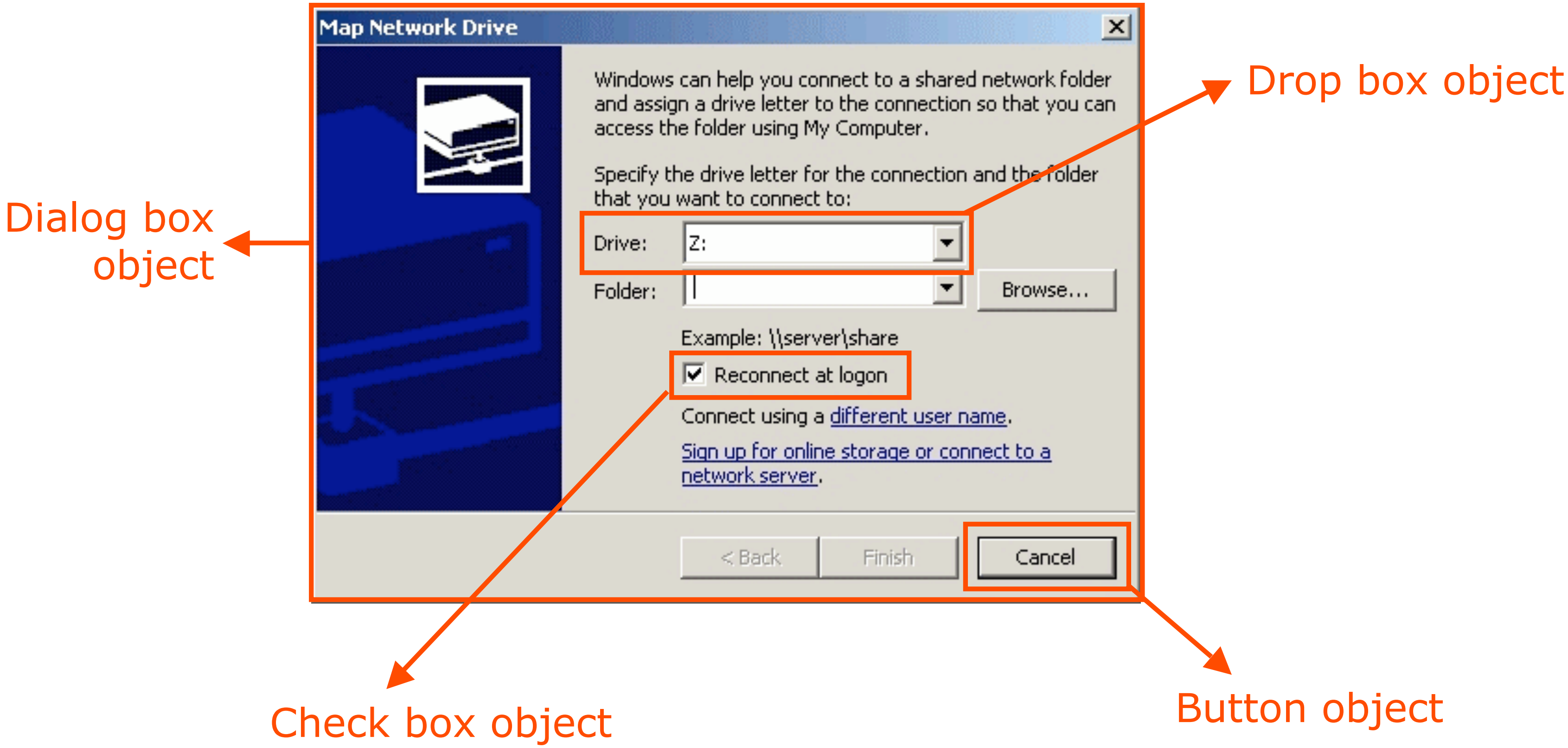
What are objects

- ‘Software objects’ are often found naturally in real-life problems
- Object oriented programming → Finding these objects and their role in your problem



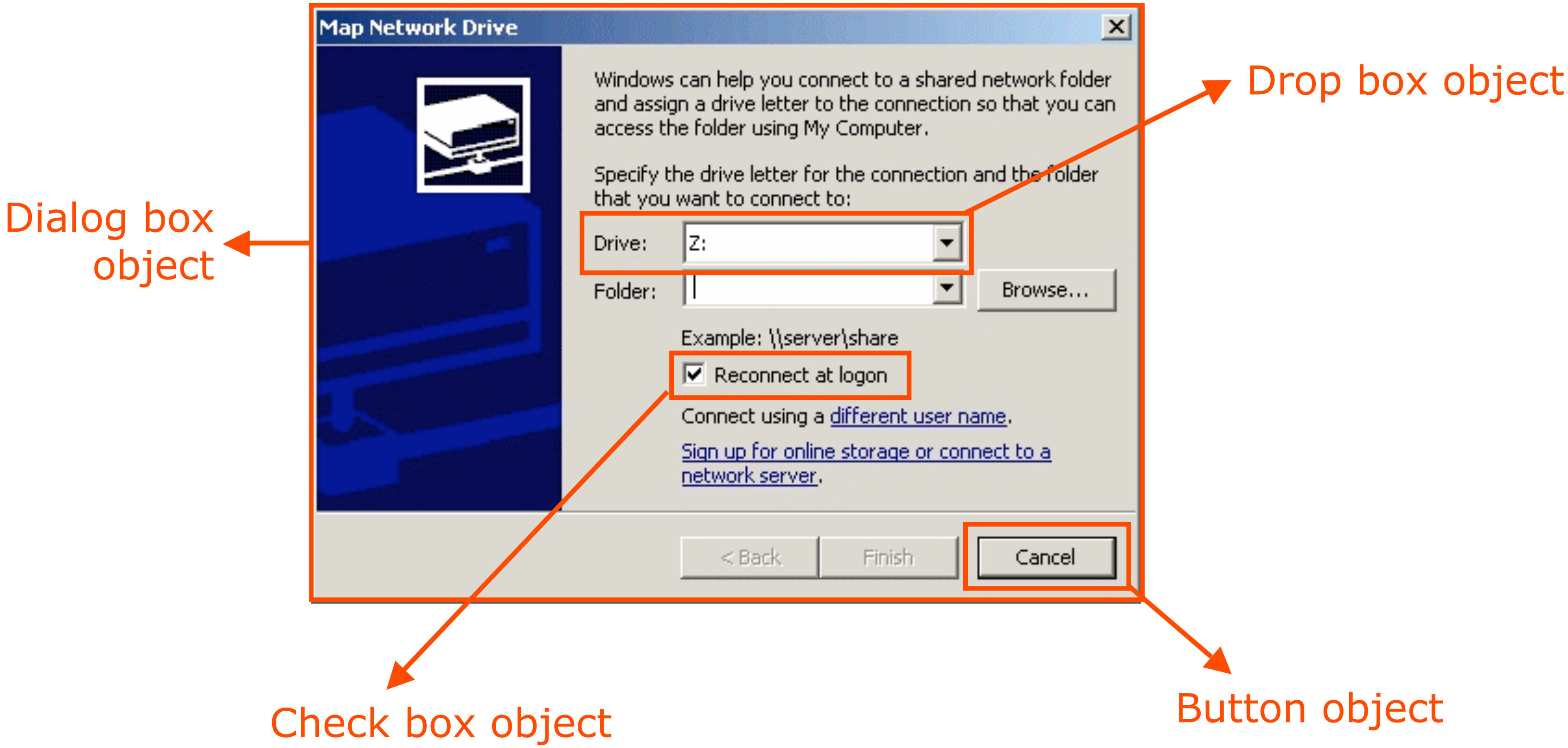
What are objects

- An object has
  - Properties : position, shape, text label
  - Behavior : if you click on the 'Cancel button' a defined action occurs



Relating objects

- Object-Oriented Analysis and Design seeks the relation between objects
  - 'Is-A' relationship (a PushButton Is-A ClickableObject)
  - 'Has-A' relationship (a DialogBox Has-A CheckBox)



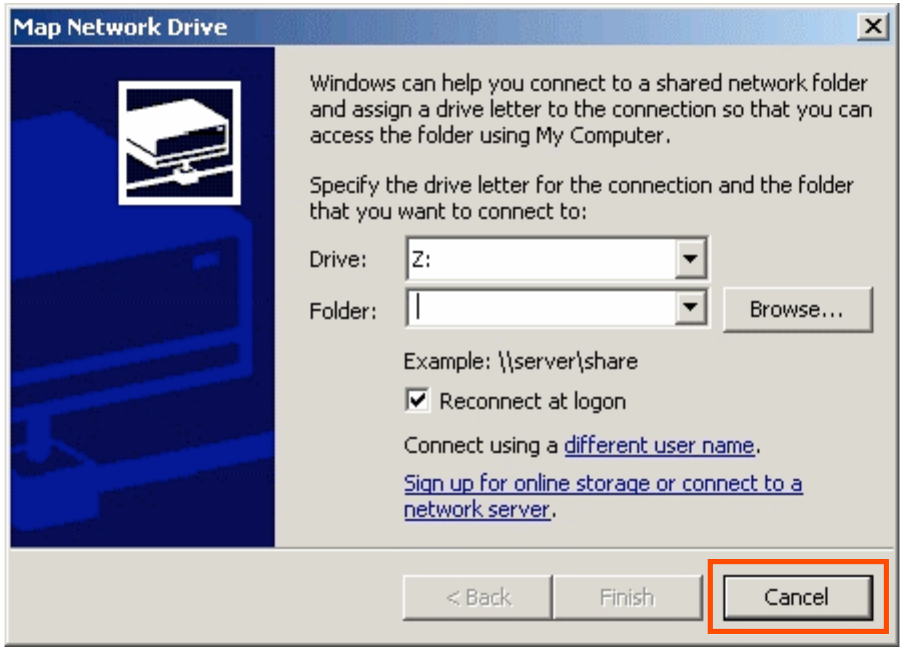
Benefits of Object-Oriented programming

- Benefits of Object-oriented programming
  - Reuse of existing code – objects can represent generic problems
  - Improved maintainability – objects are more self contained that 'subroutines' so code is less entangled
  - Often a 'natural' way to describe a system – see preceding example of dialog box
- But…
  - Object oriented modeling does not substitute for sound thinking
  - OO programming does not *guarantee* high performance, but it doesn't stand in its way either
- Nevertheless
  - *OO programming is currently the best way we know to describe complex systems*

Basic concept of OOAD

- Object-oriented programming revolves around *abstraction* of your problem.
  - Separate *what you do* from *how you do it*

- *Example – PushButton object*



PushButton is a complicated piece of software – Handling of mouse input, drawing of graphics etc..

Nevertheless you can use a PushButton object and don't need to know anything about that. Its public interface can be very simple: My name is 'cancel' and I will call function `doTheCancel()` when I get clicked

Techniques to achieve abstraction

- Abstraction is achieved through

1. **Modularity**
2. **Encapsulation**
3. **Inheritance**
4. **Polymorphism**

Modularity

- Decompose your problem logically in independent units
  - Minimize dependencies between units – **Loose coupling**
  - Group things together that have logical connection – **Strong cohesion**

- Example
  - Grouping actions and properties of a bank account together

```
long getBalance()
void print()
void calculateInterest()

char* ownersName
long   accountNumber
long   accountBalance
```

Account

Encapsulation

- Separate interface and implementation and shield implementation from object ‘users’

```
long getBalance()
void print()
void calculateInterest()
```

interface

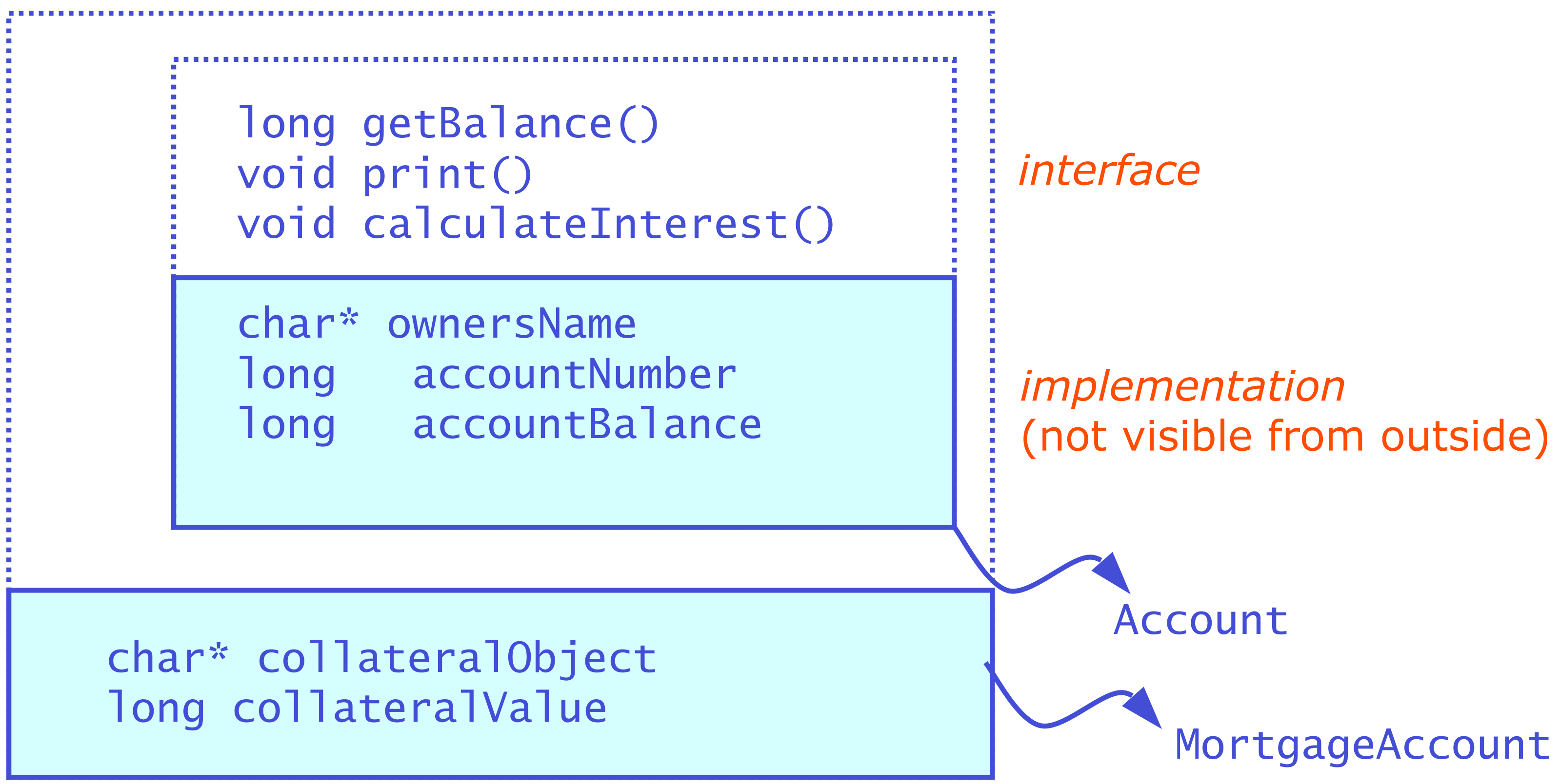
```
char* ownersName
long   accountNumber
long   accountBalance
```

implementation
(not visible from outside)

Account

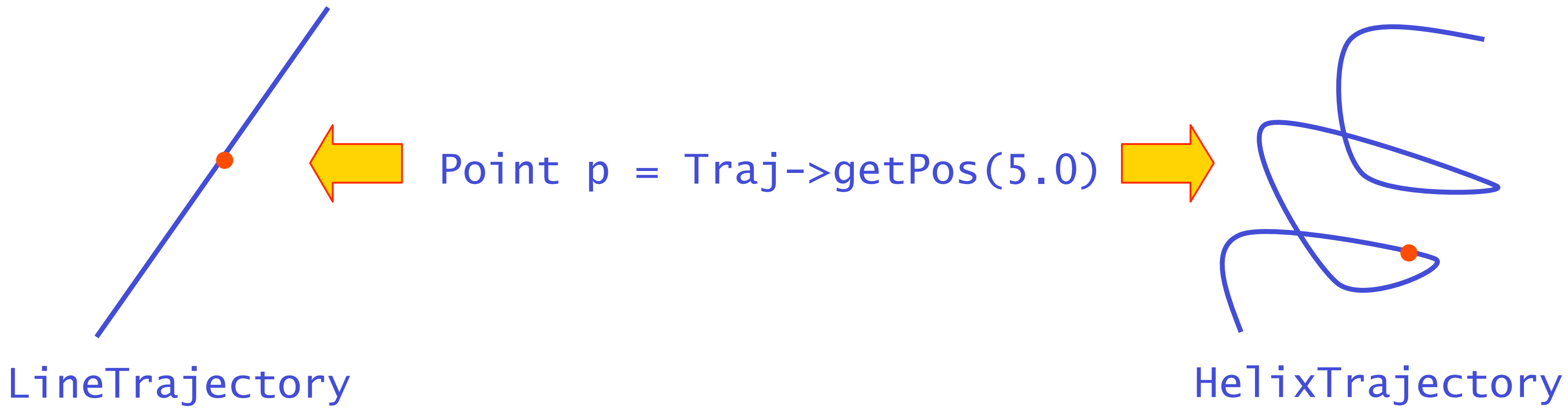
Inheritance

- Describe new objects in terms of existing objects
- Example of mortgage account



Polymorphism

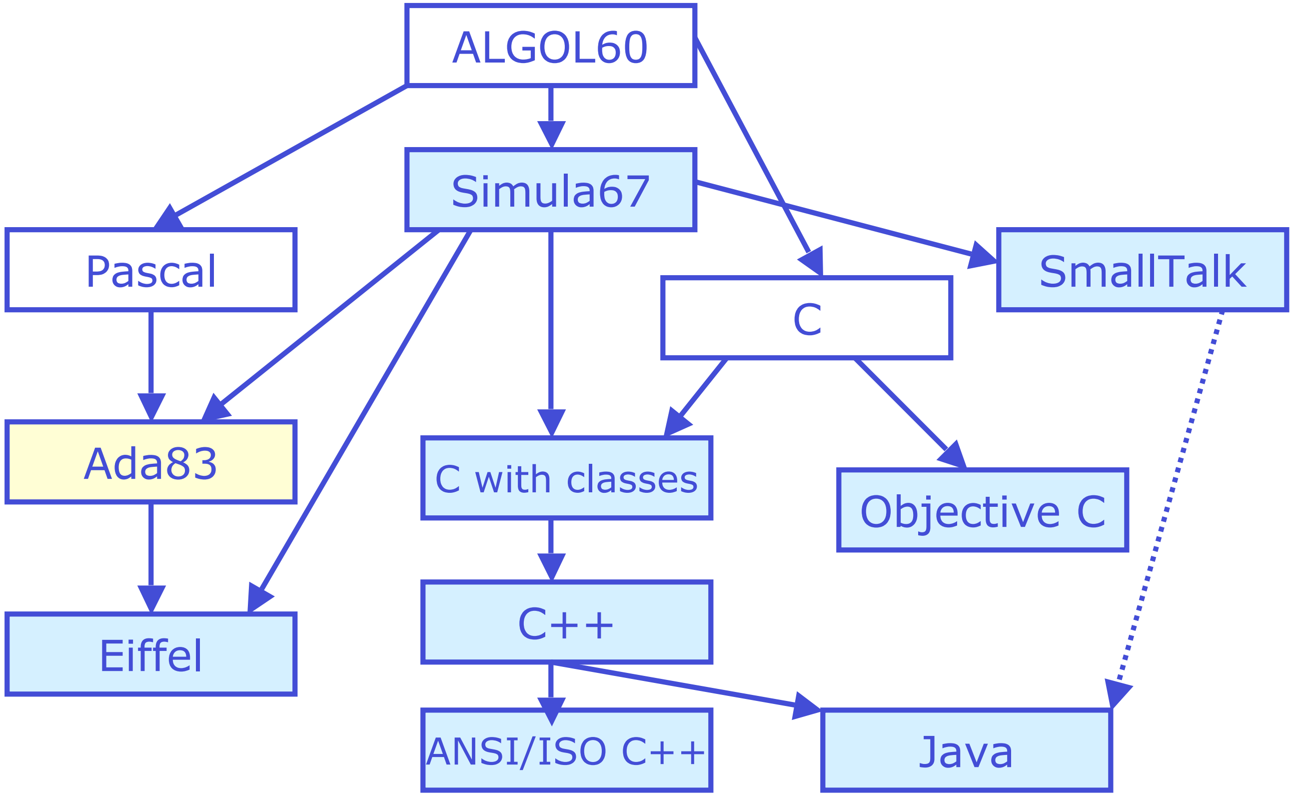
- Polymorphism is the ability to treat objects of different types the same way
  - You don’t know exactly what object you’re dealing with but you know that you can interact with it through a standardized interface
  - Requires some function call decisions to be taken at run time
- Example with trajectories
  - Retrieve position at a flight length of 5 cm
  - Same interface works for different objects with identical interface



Introduction to C++

- Wide choice of OO-languages – why program in C++?
  - It depends on what you need…
- Advantage of C++ – It is a compiled language
  - When used right the fastest of all OO languages
  - Because OO techniques in C++ are resolved and implemented at compile time rather than runtime so
    - **Maximizes run-time performance**
    - **You don't pay for what you don't use**
- Disadvantage of C++ – syntax more complex
  - Also, realizing performance advantage not always trivial
- C++ best used for large scale projects where performance matters
  - C++ rapidly becoming standard in High Energy Physics for mainstream data processing, online data acquisition etc…
  - Nevertheless, If your program code will be O(100) lines and performance is not critical C, Python, Java may be more efficient

C++ and other programming languages



- NB: Java very similar to C++, but simpler
  - Simpler syntax as all OO support is implemented at run-time
  - If you know C++ Java will be easy to learn

Outline of the course

1. Introduction and overview
2. Basics of C++
3. Modularity and Encapsulation – Files and Functions
4. Class Basics
5. Object Analysis and Design
6. The Standard Library I – Using IOstreams
7. Generic Programming – Templates
8. The Standard Library II – The template library
9. Object Orientation – Inheritance & Polymorphism
10. Robust programming – Exception handling
11. Where to go from here

1 The basics of C++

“Hello world” in C++

- Let start with a very simple C++ program

```
// my first program in C++
#include <iostream>

int main () {
  std::cout << "Hello World!“ << std::endl;
  return 0;
}
```

“Hello world” in C++

- Let start with a very simple C++ program

```
// my first program in C++
#include <iostream>

int main () {
  std::cout << "Hello Worl        < std::endl;
  return 0;
}
```

Anything on line after // in C++ is considered a comment

“Hello world” in C++

- Let start with a very simple C++ program

```
// my first program in C++
#include <iostream>

int main () {
  std::cout << "He
  return 0;
}
```

Lines starting with # are directives for the preprocessor

Here we include some standard function and type declarations of objects defined by the ‘iostream’ library

- The preprocessor of a C(++) compiler processes the source code before it is passed the compiler. It can
  - Include other source files (using the #include directive)
  - Define and substitute symbolic names (using the #define directive)
  - Conditionally include source code (using the #ifdef, #else, #endif directives)

"Hello world" in C++

- Let start with a very simple C++ program

```
// my first program in C++
#include <iostream>

int main () {
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

Beginning of the main() function declaration.

- The main() function is the default function where all C++ program begin their execution.
  - In this case the main function takes no input arguments and returns an integer value
  - You can also declare the main function to take arguments which will be filled with the command line options given to the program

“Hello word” in C++

- Let start with a very simple C++ program

```
// my first program i
#include <iostream>

int main () {
  std::cout << "Hello World!“ << std::endl;
  return 0;
}
```

Use iostream library objects to print string to standard output

- The names **`std::cout`** and **`std::endl`** are declared in the ‘header file’ included through the ‘#include <iostream>’ preprocessor directive.
- The std::endl directive represents the ‘carriage return / line feed’ operation on the terminal

“Hello word” in C++

- Let start with a very simple C++ program

```
// my first program in C++
#include <iostream>

int main () {
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

The return statement passed the return value back to the calling function

- The return value of the main() function is passed back to the operating system as the ‘process exit code’

Compiling and running 'Hello World'

- Example using Linux, (t)csh and g++ compiler

```
unix> g++ -o hello hello.cc

unix> hello
Hello World!

unix> echo $status
0
```

Convert c++ source code into executable

Run executable 'hello'

Print exit code of last run process (=hello)

Outline of this section

- Jumping in: the 'hello world' application

- Review of the basics
  - **Built-in data types**
  - **Operators on built-in types**
  - **Control flow constructs**
  - **More on block {} structures**
  - Dynamic Memory allocation

```
int main() {
   int a   = 3 ;
   float b = 5 ;

   float c = a * b + 5 ;

   if ( c > 10) {
      return 1 ;
   }

   return 0 ;
}
```

Review of the basics – built-in data types

- C++ has only few built-in data types

type name	type description
`char`	ASCII character, 1 byte
`int`, `signed int`, `unsigned int`, `short int`, `long int`	Integer. Can be signed, unsigned, long or short. Size varies and depends on CPU architecture (2,4,8 bytes)
`float`, `double`	Floating point number, single and double precision
`bool`	Boolean, can be true or false (1 byte)
`enum`	Integer with limited set of named states `enum fruit { apple,pear,citrus }`, or `enum fruit { apple=0,pear=1,citrus}`

- More complex types are available in the 'Standard Library'
  - A standard collection of tools that is available with every compiler
  - But these types are not fundamental as they're implement using standard C++
  - We will get to this soon

Defining data objects – variables

- Defining a data object can be done in several ways

```
int main() {
   int j ;     // definition – initial value undefined
   int k = 0 ; // definition with assignment initialization
   int l(0) ;  // definition with constructor initialization

   int m = k + l // initializer can be any valid C++ expressions

   int a,b=0,c(b+5) // multiple declaration – a,b,c all integers
}
```

- Data objects declared can also be declared constant

```
int main() {
   const float pi = 3.14159268 ; // constant data object
   pi = 2 ; // ERROR – doesn't compile
}
```

Defining data objects – variables

- Const variables must be initialized

```
int main() {
   const float pi ;       // ERROR – forgot to initialize

   const float e = 2.72; // OK
   const float f =  5*e ;// OK – expression is constant
}
```

- Definition can occur at any place in code

```
int main() {
   float pi = 3.14159268 ;
   cout << “pi = “ << pi << endl ;

   float result = 0; // ‘floating’ declaration OK
   result = doCalculation() ;
}
```

  – Style tip: always declare variables as close as possible to point of first use

Literal constants for built-in types

- Literal constants for integer types

```
int j = 16     // decimal
int j = 0xF    // hexadecimal (leading 0x)
int j = 020    // octal (leading 0)

unsigned int k = 4294967280U // unsigned literal
```

  - Hex, octal literals good for bit patterns
    (hex digit = 4 bits, octal digit = 3 bits)
  - Unsigned literals good for numbers that are
    too large for signed integers
    (e.g. between 2^32/2 and 2^32-1)

- Literal constants for character types

```
char ch = 'A'  // Use single quotes;
```

  - Escape sequences exist for special characters

\n	newline
\r	carriage return
\t	tabulation
\v	vertical tabulation
\b	backspace
\f	page feed
\a	alert (beep)
\'	single quotes (')
\"	double quotes (")
\?	question (?)
****	inverted slash (\)

Arrays

- C++ supports 1-dimensional and N-dimensional arrays
  - Definition

    ```
    Type name[size] ;
    Type name[size1][size2]…[sizeN] ;
    ```

  - Array dimensions in definition must be constants

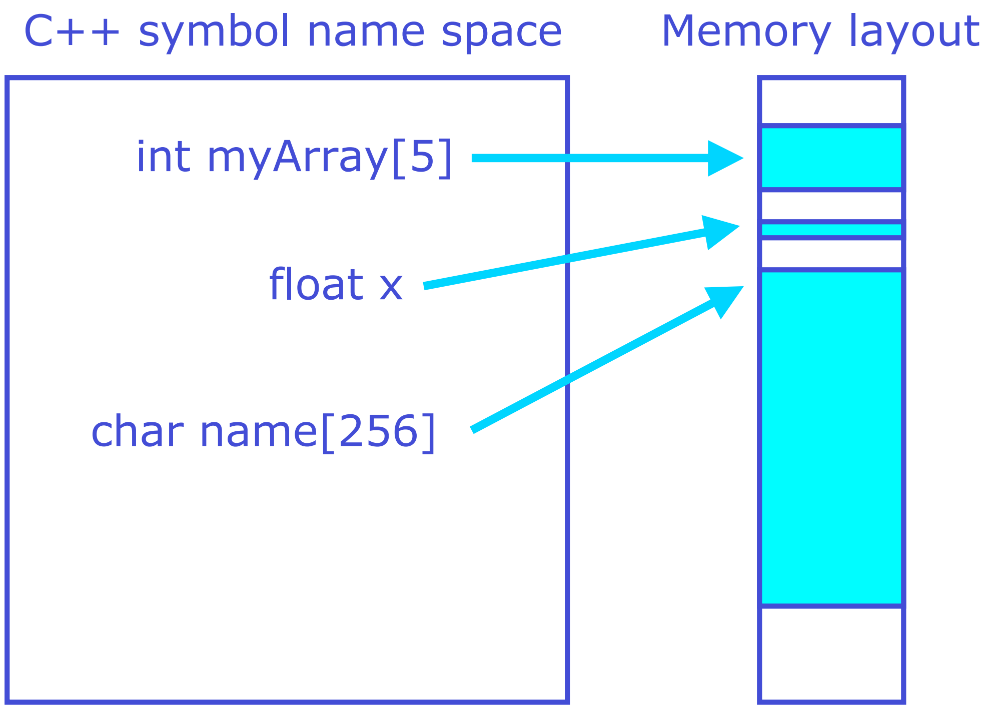
    ```
    float x[3] ;     // OK

    const int n=3 ;
    float x[n] ;     // OK

    int k=5 ;
    float x[k] ;     // ERROR!
    ```

  - *First element is always **0***
  - Assignment initialization possible

    ```
    float x[3]     = { 0.0, 5.7 , 2.3 } ;
    float y[2][2]  = { 0.0, 1.0, 2.0, 3.0 } ;
    float y[3]     = { 1.0 } ; // Incomplete initialization OK
    ```

Declaration versus definition of data

- Important fine point: definition of a variable is two actions
    1. Allocation of memory for object
    2. Assigning a symbolic name to that memory space



    - C++ symbolic name is a way for programs to give understandable names to segments of memory.
    - But it is an artifact: no longer exists once the program is compiled

References

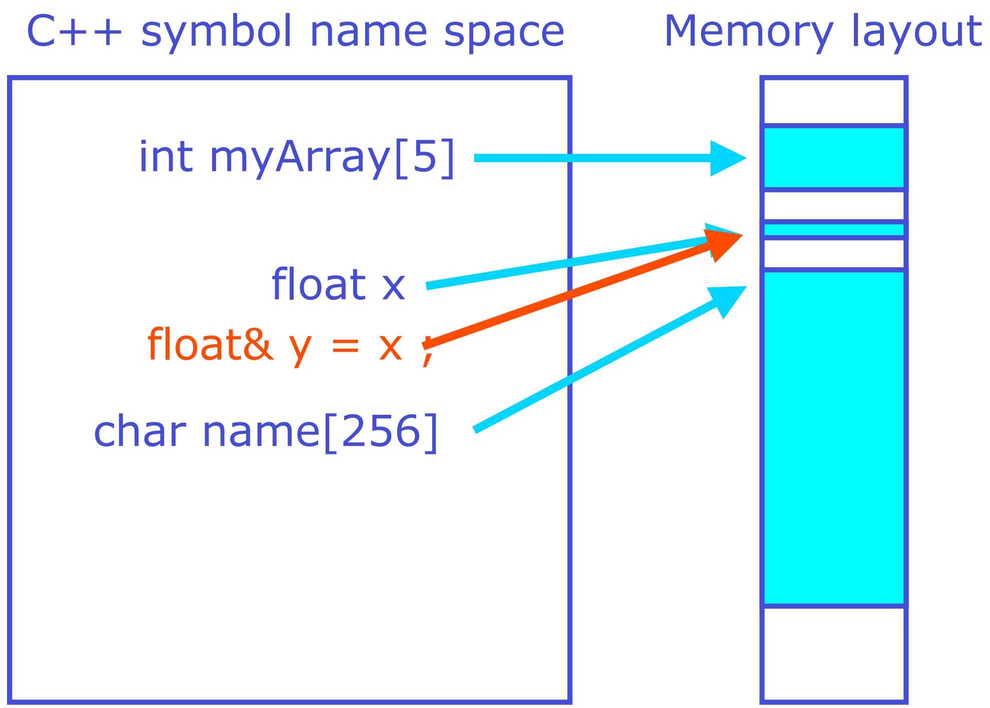
- C++ allows to create 'alias names', a different symbolic name references an already allocated data object
  - Syntax: '`Type& name = othername`'
  - References do not necessarily allocate memory
- Example

```
int x ;       // Allocation of memory for in
              // and declaration of name 'x'
int& y = x ;  // Declaration of alias name 'y'
              // for memory refenced by 'x'

x = 3 ;
cout << x << endl ; // prints '3'
cout << y << endl ; // also prints '3'
```

  - Concept of references will become more interesting when we'll talk about functions

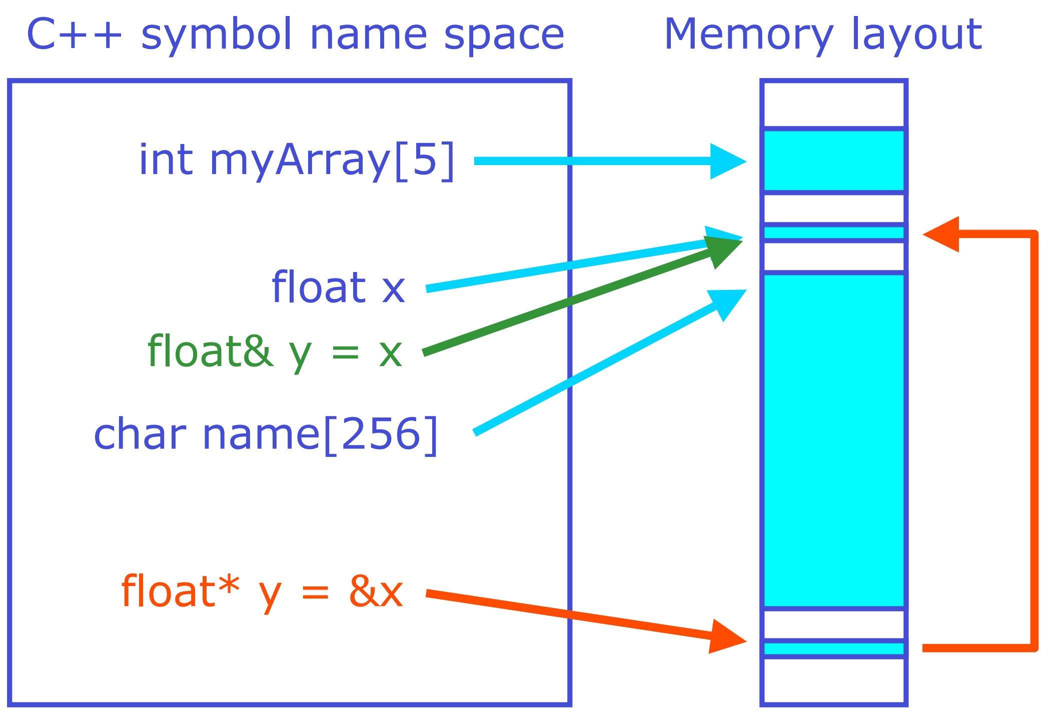
References

- Illustration C++ of reference concept
  - Reference is symbolic name that points to same memory as initializer symbol



Pointers

- Pointers is a variable that contains a memory address
  - Somewhat similar to a reference in functionality, but fundamentally different in nature: a pointer is always an object in memory itself
  - Definition: '**`TYPE* name`**' makes pointer to data of type TYPE



Pointers

- Working with pointers
  - Operator & takes memory address of symbol object (=pointer value)
  - Operator * turns memory address (=pointer value) into symbol object
- Creating and reading through pointers

```
int x = 3, y = 4 ;
int* px  ;             // allocate px of type 'pointer to integer'
px = &x ;              // assign 'memory address of x' to pointer px

cout << px << endl ; // Prints 0x3564353, memory address of x
cout << *px << e`ndl ;// Prints 3, value of x, object pointed to by px
```

- Modifying pointers and objects pointed to

```
*px = 5 ;              // Change value of object pointed to by px (=x) ;
cout << x << endl      // Prints 5 (since changed through px)
px = &y ;              // Reseat pointer to point to symbol named 'y'

cout << px << endl ; // Prints 0x4863813, memory address of y
cout << *px << endl ;// Prints 4, value of y, objected pointed to by px
```

Pointers continued

- Pointers are also fundamentally related to arrays

```
int a[3]  = { 1,2,3} ; // Allocates  array of 3 integers
int* pa   = &a[0] ;    // Pointer pa now points a[0] ;

cout << *pa << endl ;      // Prints '1'
cout << *(pa+1) << endl ; // Prints '2'
```

- Pointer (pa+1) points to next element of an array
  - This works regardless of the type in the array
  - In fact **`a`** itself is a pointer of type **`int*`** pointing to **`a[0]`**

- The **Basic Rule** for arrays and pointers
  - **`a[i]` is equivalent to `*(a+i)`**

Pointers and arrays of char – strings

- Some special facilities exists for arrays of `char`
  - char[] holds strings and is therefore most commonly used array

- Initialization of character arrays:
  - String literals in double quotes are of type ‘char *’, i.e.

```
const char* blah = “query”
```

    is equivalent to

```
const char tmp[7] = {‘q’,’w’,’e’,’r’,’t’,’y’,0}’ ;
const char* blah = tmp ;
```

  - Recap: single quoted for a single char, double quotes for a const pointer to an array of chars

- Termination of character arrays
  - Character arrays are by convention ended with a null char (\0)
  - Can detect end of string without access to original definition
    - For example for strings returned by “a literation expression”

Strings and string manipulation

- Since char[] strings are such an common objects
  - the 'Standard Library' provides some convenient manipulation functions

- Most popular char[] manipulation functions

```
//  Length of string
int strlen(const char* str) ;

// Append str2 to str1 (make sure yourself str1 is large enough)
char* strcat(char* str1, const char* str2) ;

// Compares strings, returns 0 if strings are identical
strcmp(const char* str1, const char* str2) ;
```

- Tip: Standard Library also provides 'class string' with superior handling
  - We'll cover class string later
  - But still need 'const char*' to interact with operating system function calls (open file, close file, etc)

Reading vs. Writing – LValues and RValues

- C++ has two important concepts to distinguish read-only objects and writeable objects
  - An **LValue** is writable and can appear on the left-hand side of an assignment operation
  - An **RValue** is read-only and may only appear on the right-hand side of assignment operations

- Example

```
int i;
char buf[10] ;

i = 5 ; // OK, i is an lvalue
5 = i ; // ERROR, 5 is not an lvalue
        // (it has no memory location)

buf[0] = ‘c’ ;    // OK buf[0] is an lvalue
buf = “qwerty” ; // ERROR, buf is immutably tied to char[10]
```

Operators and expressions – arithmetic operators

- Arithmetic operators overview

Name	Operator
Unary minus	`-x`
Multiplication	`x * y`
Division	`x / y`

Name	Operator
Modulus	`x % y`
Addition	`x + y`
Subtraction	`x - y`

- Arithmetic operators are evaluated from **left to right**
  - `40 / 4 * 5 = (40 / 4) * 5 = 50` (not 2)

- In case of mixed-type expressions compiler automatically converts integers up to floats

```
int i = 3, j = 5 ;
float x = 1.5 ;

float y = i*x ; // = 4.5 ; int i promoted to float
float z = j/i ; // = 1.0 ; '/' has precedence over '='
```

Operators and expressions – increment/decrement operators

- In/Decrement operators

Name	Operator
Prefix increment	`++x`
Postfix increment	`x++`
Prefix decrement	`--x`
Postfix decrement	`x--`

- Note difference
  - ***Prefix*** operators return value ***after*** operation
  - ***Postfix*** operators return value ***before*** operation
- Examples

```
int x=0 ;
cout << x++ << endl ; // Prints 0
cout << x << endl ;   // Prints 1

cout << ++x << endl ; // Prints 2
cout << x << endl ;   // Prints 2
```

Operators and expressions – relational operators

- Relational operators

Name	Operator
Less than	`x < y`
Less than or equal to	`x <= y`
Greater than or equal to	`x >= y`
Greater than	`x > y`
Equal to	`x == y`
Not equal to	`x != y`

- All relational operators yield `bool` results
- Operators `==`, `!=` have precedence over `<`,`<=`,`>=`,`>`

Operators and expressions – Logical operators

- Logical operators

Name	Operator
Logical NOT	`!x`
Logical AND	`x>3 && x<5`
Logical OR	`x==3 || x==5`

Do not confuse with bit-wise AND (&) and bit-wise OR (|)

- All logical operators take `bool` arguments and return `bool`
  - If input is not `bool` it is *converted* to `bool`
  - Zero of any type maps to `false`, anything else maps to `true`
- Logical operators are evaluated from left to right
  - *Evaluation is **guaranteed to stop** as soon as outcome is determined*

```
float x, y ;
…
if (y!=0. && x/y < 5.2) // safe against divide by zero
```

Operators and expressions – Bitwise operators

- Bitwise operators

Name	Operator	Example
Bitwise complement	`~x`	0011000 → 1100111
Left shift	`x << 2`	000001 → 000100
Right shift	`x >> 3`	111111 → 000111
Bitwise AND	`x & y`	1100 & 0101 = 0100
Bitwise OR	`x | y`	1100 | 0101 = 1101
Bitwise XOR	`x ^ y`	1100 ^ 0101 = 1001

- Remarks
  - Bitwise operators cannot be applied to floating point types
  - Mostly used in online, DAQ applications where memory is limited and ‘bit packing is common’
  - Do not confuse logical or, and (||,&&) with bitwise or, and (|,&)

Operators and expressions – Assignment operators

- Assignment operators

Name	Operator
Assignment	x = 5
Addition update	x += 5
Subtraction update	x -= 5
Multiplication update	x *= 5
Division update	x /= 5
Modulus update	x %= 5
Left shift update	x <<= 5
Right shift update	x >>= 5
Bitwise AND update	x &= 5
Bitwise OR update	x |= 5
Bitwise XOR update	x ^= 5

Operators and expressions – Assignment operators

- Important details on assignment operators
  - **Left-hand** arguments must be **lvalues** (naturally)
  - Assignment is evaluated **right to left**
  - Assignment operator **returns left-hand** value of expression

- Return value property of assignment has important consequences
  - **Chain** assignment is possible!

```
x = y = z = 5  // OK! x = ( y = ( z = 5 ))
               //     x = ( y = 5)
               //     x = 5
```

  - **Inline** assignment is possible

```
int x[5], i ;
x[i=2] = 3 ; // i is set to 2, x[2] is set to 3
```

Operators and expressions – Miscellaneous

- Inline conditional expression: the ternary `?:` operator
  - Executes inline if-then-else conditional expression

```
int x = 4 ;
cout << ( x==4 ? "A" : "B" ) << endl ; // prints "A" ;
```

- The comma operator (`expr1, expr2, expr3)`
  - Evaluates expressions sequentially, returns *rightmost* expression

```
int i=0, j=1, k=2 ;
cout<< (i=5, j=5, k) <<endl ; // Prints '2', but i,j set to 5
```

- The `sizeof` operator
  - Returns size in bytes of operand, argument can be *type* or *symbol*

```
int size1 = sizeof(int) ; // =4 (on most 32-bit archs)

double x[10]
int size2 = sizeof(x) ;   // =10*sizeof(double)
```

Conversion operators

- Automatic conversion
  - All type conversion that can be done 'legally' and without loss of information are done automatically
  - Example: float to double conversion

```
float f = 5 ;
double d = f ; // Automatic conversion occurs here
```

- Non-trivial conversions are also possible, but not automatic
  - Example: `float` to `int`, `signed int` to `unsigned int`
  - If conversion is non-trivial, conversion is not automatic → you must it request it with a ***conversion operator***

- C++ has a variety of ways to accomplish conversions
  - C++ term for type conversion is '**cast**'
  - Will focus on 'modern' methods and ignore 'heritage' methods

Conversion operators – Explicit casts

- For conversions that are ‘legal’ but may result in truncation, loss of precision etc…: **static_cast**

```
float f = 3.1
int i = static_cast<int>(f) ; // OK, i=3 (loss of precision)
int* i = static_cast<int*>(f) ; // ERROR float != pointer
```

- For conversions from ‘const X’ to ‘X’, i.e. to override a logical const declaration: **const_cast**

```
float f = 3.1 ;
const float& g = f ;
g = 5.3 ;                            // ERROR not allowed, g is const
float& h = const_cast<float&>(g) ; // OK g and h of same type
h = 5.3 ;                            // OK, h is not const
```

Conversion operators – Explicit casts

- Your last resort: **reinterpret_cast**

```
float* f ;
int* i = reintepret_cast<int*>(f) ; // OK, but you take
                  // responsibility for the ensuing mess…
```

- You may need more than one cast to do your conversion

```
const float f = 3.1
int i = static_cast<int>(f) ; // ERROR static_cast cannot
                              // convert const into non-const

const float f = 3.1
int i = static_cast<int>( const_cast<float>(f) ) ; // OK
```

  – It may look verbose it helps you to understand your code as all aspects of the conversion are explicitly spelled out

Control flow constructs – `if/else`

- The `if` construct has three formats
  - Parentheses around expression required
  - Brackets optional if there is only one statement (but put them anyway)

```
if (expr) {
   statements ; // evaluated if expr is true
}
```

```
if (expr) {
  statements ; // evaluated if expr is true
} else {
  statements ; // evaluated if expr is false
}
```

```
if (expr1) {
  statements ;     // evaluated if expr1 is true
} else if (expr2) {
  statements ;     // evaluated if expr2 is true
} else {
  statements ;     // evaluated if neither expr is true
}
```

Intermezzo – coding style

- C++ is free-form so there are no rules
- But style matters for readability, some suggestions
  - One statement per line
  - **Always put {} brackets** even if statement is single line
  - Common indentation styles for {} blocks

```
if (foo==bar) {
    statements ;
} else {
    statements ;
}
```

```
if (foo==bar)
{
    statements ;
}
else
{
    statements ;
}
```

Try to learn yourself this style,
it is more compact and more
readable (especially when you're
more experienced)

Control flow constructs – while

- The `while` construct

```
while (expression) {
  statements ;
}
```

  - Statements will be executed if expression is true
  - At end, expression is re-evaluated. If again true, statements are again executed

- The `do/while` construct

```
do {
  statements ;
} while (expression) ;
```

  - Similar to **`while`** construct except that statements are always executed once before expression is evaluated for the first time

Control flow constructs – for

- The for construct

```
for (expression1 ; expression2 ; expression3 ) {
  statements ;
}
```

  - is equivalent to

```
expression1 ;
while (expression2) {
  statements ;
  expression3 ;
}
```

- Most common looping construct

```
int i ;
for (i=0 ; i<5 ; i++) {
  // Executes with i=0,1,2,3 and 4
}
```

Control flow constructs – for

- Expressions may be empty

```
for (;;) {
  cout << “Forever more” << endl ;
}
```

- Comma operator can be useful to combine multiple operations in expressions

```
int i,j
for (i=0,j=0 ; i<3 ; i++,j+=2) {
  // execute with i=0,j=0, i=1,j=2, i=2,j=4
}
```

Control flow constructs – break and continue

- Sometimes you need to stop iterating a `do`, `do/while` or `for` loop prematurely
  - Use `break` and `continue` statements to modify control flow

- The `break` statement
  - Terminate loop construct *immediately*

```
int i = 3 ;
while(true) { // no scheduled exit from loop
  i -= 1 ;
  if (i<0) break ; // exit loop
  cout << i << endl ;
}
```

  - Example prints ‘2’, ’1’ and ’0’. Print statement for i=-1 never executed

Control flow constructs – break and continue

- The `continue` statement
  - Continue stops execution of loops statements and *returns to evaluation of conditional expression*

```
char buf[12] = "abc,def,ghi"
for (int i=0 ; i<12 ; i++) {
  if (buf[i]==',') continue ; // return to for()
                              // if ',' is encountered
  cout << buf[i]  ;
}
cout << endl ;
```

  - Output of example `'abcdefghi'`
  - Do not confuse with FORTRAN 'continue' statement -- Very different meaning!

- Both `break` and `continue` only affect the *innermost* loop
  - When you are using nested loops

Control flow constructs -- switch

- The switch construct

```
switch (expr) {
  case constant1:
    statements ; // Evaluated if expr==const1
    break ;

  case constant2:
  case constant3:
    statements ; // Evaluated if expr==const2 or const3
    break ;

  default:
    statements ; // Evaluated expression matched none
    break ;
}
```

  - Most useful for decision tree algorithms

- If break is omitted execution continues with next case evaluation
  - Usually you don't want this, so watch the breaks

Control flow constructs – switch (example)

- `switch` works very elegantly with `enum` types
  - `enum` naturally has finite set of states
- case expressions must be constant but can be any valid expression
  - Example:

```
enum color { red=1, green=2, blue=4 };
color paint = getcolor() ;
switch (paint) {
  case red:
  case green:
  case blue:
    cout << “primary color” << endl ;
    break ;

  case red+green:
    cout << “yellow” << end;
    break ;

  case red+blue:
    cout << “magenta” << end;
    break ;

  case blue+green:
    cout << “cyan” << end;
    break ;

  default:
    cout << “white” << endl ;
    break ;
}
```

Some details on the block {} statements

- Be sure to understand all consequences of a block {}
  - The lifetime of automatic variables inside the block is limited to the end of the block (i.e up to the point where the } is encountered)

```
main() {
  int i = 1 ;

  if (x>0) {            <-- Memory for
    int i = 0 ;             'int i' allocated
    // code
  } else {     Memory for 'int i' released -->
    // code
  }
}
```

  - A block introduces a new scope : it is a separate name space in which you can define new symbols, even if those names already existed in the enclosing block

Scope – more symbol visibility in {} blocks

- Basic C++ scope rules for variable definitions
  - In given location all variables defined in local scope are visible
  - All variables defined in enclosing scopes are visible
  - Global variables are always visible
  - Example

```
int a ;
int main() {
   int b=0 ;

   if (b==0) {
      int c = 1;
   }

}
```

a, b visible

a, b, c visible

Scoping rules – hiding

- What happens if two variables declared in different scopes have the same name?
  - Definition in inner scope ***hides*** definition in outer scope
  - It is legal to have two variables with the same name defined in different scopes

```
int a ;
int main() {
   int b=0 ;              ← 'b' declared in main() visible

   if (b==0) {
LEGAL! → int b = 1;       ← 'b' declared in if() visible
                            'b' declared in main() hidden!
   }

}
```

  - NB: It is not legal to have two definitions of the same name in the same scope, e.g.

```
int main() {
int b ;
…
int b ; ERROR!
}
```

Scoping rules – The :: operator

- Global variables, even if hidden, can always be accessed using ***the scope resolution operator ::***

```
int a=1 ;

int main() {
   int a=0 ;    ← LEGAL, but hides global 'a'

   ::a = 2 ;    ← Changes global 'a'
}
```

- No tools to resolve symbols from intermediate unnamed scope
  - Solution will be to use 'named' scopes: namespaces or classes
  - More on classes later

More on memory use

- By default all objects defined *outside* {} blocks (global objects) are allocated statically
  - Memory allocated before execution of `main()` begins
  - Memory released after `main()` terminates

- By default all defined objects defined *inside* {} blocks are 'automatic' variables
  - Memory allocated when definition occurs
  - Memory released when closing bracket of scope is encountered

```
if (x>0) {
    int i = 0 ;     <---- Memory for 'int i' allocated
    // code
}                   <---- Memory for 'int i' released
```

  - You can override behavior of variables declared in {} blocks to be statically allocated using the `static` keyword

More on memory allocation

- Example of static declaration

```
void func(int i_new) {
   static int i = 0 ;
   cout << “old value = “ << i << endl ;
   i = i_new ;
   cout << “new value = “ << i << endl ;
}

main() {
  func(1) ;
  func(2) ;
}
```

  - Output of example

```
old value = 0 ;
new value = 1 ;
old value = 1 ;
new value = 2 ;
```

Value of `static int i` preserved between `func()` calls

Dynamic memory allocation

- Allocating memory at run-time
  - When you design programs you cannot always determine how much memory you need
  - You can allocate objects of size unknown at compile time using the 'free store' of the C++ run time environment

- Basic syntax of runtime memory allocation
  - Operator `new` allocates single object, returns pointer
  - Operator `new[]` allocates array of objects, returns pointer

```
// Single objects
Type* ptr = new Type ;
Type* ptr = new Type(initValue) ;

// Arrays of objects
Type* ptr = new Type[size] ;
Type* ptr = new Type[size][size]…[sizeN] ;
```

Releasing dynamic memory allocation

- Operator delete releases dynamic memory allocated with new

```
// Single objects
delete ptr ;

// Arrays of objects
delete[] ptr ;
```

  - **Be sure to use delete[] for allocated arrays**. A mismatch will result in an incomplete memory release

- How much memory is available in the free store?
  - As much as the operating system lets you have
  - If you ask for more than is available your program will terminate in the new operator
  - It is possible to intercept this condition and continue the program using ‘exception handling’ (we’ll discuss this later)

Dynamic memory and leaks

- A common problem in programs are memory leaks
  - Memory is allocated but never released even when it is not used anymore
  - Example of leaking code

```
void leakFunc() {
  int* array = new int[1000] ;
  // do stuff with array
}

int main() {
   int i ;
   for (i=0 ; i<1000 ; i++) {
     leakFunc() ; // we leak 4K at every call
  }
}
```

Leak happens right here
we loose the pointer `array` *here and with that our only possibility to release memory in future*

Dynamic memory and leaks

- Another scenario to leak memory
  - Misunderstanding between two functions

```
int* allocFunc() {
  int* array = new int[1000] ;
  // do stuff with array
  return array ;
}

int main() {
   int i ;
   for (i=0 ; i<1000 ; i++) {
     allocFunc() ;
   }
}
```

allocFunc() allocates memory but pointer as return value memory is not leaked yet

Author of main() doesn't know that it is supposed to delete array returned by allocFunc()

Leak occurs here, pointer to dynamically allocated memory is lost before memory is released

Dynamic memory and ownership

- Avoiding leaks is a matter of good bookkeeping
  - All memory allocated should be released again

- Memory handling logistics usually described in terms of **ownership**
  - The 'owner' of dynamically allocated memory is responsible for releasing the memory again
  - **Ownership is a 'moral concept'**, not a C++ syntax rule. Code that never releases memory it allocated is legal, but may not work well as program size will increase in an uncontrolled way over time
  - Document your memory management code in terms of ownership

Dynamic memory allocation

- Example of dynamic memory allocation with ownership semantics
  - Less confusion about division of responsabilities

```
int* makearray(int size) {
   // NOTE: caller takes ownership of memory
   int* array = new int(size) ;

   int i ;
   for (i=0 ; i<size ; i++) {
     array[i] = 0 ;
   }
}

int main() {
  // Note: We own array ;
  int* array = makearray(1000) ;

  delete[] array ;
}
```

2 Files and Functions

Introduction – Files, functions and namespaces

- Contents of this chapter
  - **Encapsulating algorithms** – functions
  - **Splitting your code into modules** – working with multiple files
  - **Decoupling symbols in modules** – namespaces
  - **Working with existing modules** – The Standard Library

Structured programming – Functions

- Functions group statements into logical units
  - Functions encapsulate algorithms

- Declaration

```
TYPE function_name(TYPE arg1, TYPE arg2, TYPE argN) ;
```

- Definition:

```
TYPE function_name(TYPE arg1, TYPE arg2, TYPE argN) {
   // body
   statements ;
   return arg ;
}
```

- Ability to declare function separate from definition important
  - Allows to separate implementation and interface
  - But also solves certain otherwise intractable problems

Forward declaration of functions

- Example of trouble using function definitions only

```
int g() {
  f() ; // g calls f – ERROR, f not known yet
}

int f() {
  g() ; // f calls g – OK g is defined
}
```

  – Reversing order of definition doesn't solve problem,
  – But **forward declaration** does solve it:

```
int f(int x) ;

int g() {
  f(x*2) ; // g calls f – OK f declared now
}

int f(int x) {
  g() ; // f calls g – OK g defined by now
}
```

Functions – recursion

- Recursive function calls are explicitly allowed in C++
  - Example

```
int factorial(int number) {

   if (number<=1) {
      return number ;
   }

   return number*factorial(number-1) ;
}
```

  - NB: Above example works only in pass-by-value implementation

- Attractive solution for inherently recursive algorithms
  - Recursive (directory) tree searches, etc…

Function arguments

- Function ***input and return*** arguments are ***both optional***
  - Function with no input arguments: `TYPE function() ;`
  - Function with no return argument: `void function(TYPE arg,…) ;`
  - Function with neither: `void function() ;`
- Pseudo type **`void`** is used as place holder when no argument is returned
- Returning a value
  - If a function is declared to return a value, a value must be return using the '`return <value>`' statement
  - The return statement may occur anywhere in the function body, but every execution path must end in a return statement

```
int func() {
   if (special_condition) {
      return -1 ;
  }
   return 0;
}
```

```
void func() {
   if (special_condition) {
      return ;
  }
   return ; // optional
}
```

  - Void functions may terminate early using '`return ;`'

Function arguments – values

- By default all functions arguments are passed by value
  - Function is passed **copies** of input arguments

```
void swap(int a, int b) ;

main() {
   int a=3, b=5 ;
   swap(a,b) ;
   cout << "a=" << a << ", b=" << b << endl ;
}

void swap(int a, int b) {
   int tmp ;
   tmp = a ;
   a = b ;
   b = tmp ;
}
// output = "a=3, b=5"
```

a and b in swap() are **copies** of a and b in main()

  - Allows function to freely modify inputs without consequences
  - Note: potentially expensive when passing large objects (arrays) by value is expensive!

Function arguments – references

- You can change this behavior by passing **references** as input arguments

```
void swap(int& a, int& b) ;

main() {
   int a=3, b=5 ;
   swap(a,b) ;
   cout << “a=“ << a << “, b=“ << b << endl ;
}

void swap(int& a, int& b) {
   int tmp ;
   tmp = a ;
   a = b ;
   b = tmp ;
}
// Output “a=5, b=3”
```

a and b in swap() are **references to** original a and b in main(). Any operation affects originals

  - Passing by reference is inexpensive, regardless of size of object
  - But allows functions to modify input arguments which may have potentially further consequences

Function arguments – const references

- Functions with 'const references' take references but promise not to change the object

```
void swap(const int& a, const int& b) {
   int tmp ;
   tmp = a ;  // OK - does not modify a
   a = b ;    // COMPILER ERROR - Not allowed
   b = tmp ;  // COMPILER ERROR - Not allowed
}
```

- Use const references instead of 'pass-by-value' when you are dealing with large objects that will not be changed
  - Low overhead (no copying of large overhead)
  - Input value remains unchanged (thanks to const promise)

Function arguments – pointers

- You can of course also pass pointers as arguments

```
void swap(int* a, int* b) ;

main() {
  int a=3, b=5 ;
  swap(&a,&b) ;
  cout << "a=" << a << ", b=" << b << endl ;
}

void swap(int* a, int* b) {
  int tmp ;
  tmp = *a ;
  *a = *b ;
  *b = tmp ;
}

// Output "a=5, b=3"
```

a and b in swap() are **pointers to** original a and b in main(). Any operation affects originals

  - Syntax more cumbersome, use references when you can, pointers only when you have to

Function arguments – references to pointers

- Passing a pointer by reference allows function to modify pointers
  - Often used to pass multiple pointer arguments to calling function

```
bool allocate(int*& a, int*& b) ;

main() {
  int* a(0), *b(0) ;
  bool ok = allocate(a,b) ;
  cout << a << " " << b << endl ;
  // prints 0x4856735 0x4927847
}

bool allocate(int*& a, int*& b) {
  a = new int[100] ;
  b = new int[100*100] ;
  return true ;
}
```

  - NB: reverse is not allowed – you can't make a pointer to a reference as a reference is not (necessarily) an object in memory

Function arguments – arrays

- Example of passing a 1D arrays as argument

```
void square(int len, int array[]) ;

main() {
   int array[3] = { 0, 1, 2 } ;
   square( sizeof(array)/sizeof(int), array) ;
   return 0 ;
}

void square (int len, int array[]) {
       while(--len>=0) {
      array[len] *= array[len] ;
  }
}
```

Remark:
Code exploits that len is a copy
Note prefix decrement use
*Note use of *= operator*

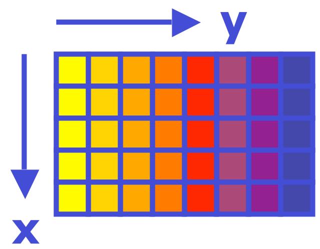
- Remember basic rule: array = pointer
  - Writing '`int* array`' is equivalent to '`int array[]`'
  - Arrays always passed 'by pointer'
  - Need to pass length of array as separate argument

Function arguments – multi-dimensional arrays

- Multi-dimensional array arguments more complicated
  - Reason: interpretation of array memory depends on dimension sizes (unlike 1D array)
- Must specify all dimensions except 1st one in declaration
  - Example

```
// arg of f declared to be N x 10 array
void f(int [][10]) ;

// Pass 5 x 10 array
int a[5][10] ;
f(a) ;

void f(int p[][10]) {
  // inside f use 2-D array as usual
     … p[i][j]…
}
```

Logical layout



Memory layout

Function arguments – char[] (strings)

- Since `char*` strings are zero terminated by convention, no separate length argument needs to be passed
  - Example

```
void print(const char* str) ;

int main(){
   const char* foo = “Hello World” ;
   print(foo) ;
   return 0 ;
}

void print (const char* str) {
   const char* ptr = str ;
   while (*ptr != 0) {
       cout << *ptr << endl ;
       ptr++ ;
   }
}
```

Function arguments – main() and the command line

- If you want to access command line arguments you can declare `main()` as follows

Array of (**char***)

```
int main(int argc, char* argv[]) {
   int i ;
   for (i=0 ; i<argc ; i++) {
      // argv[i] is 'char *'
      cout << "arg #" << i << " = " << argv[i] << endl ;
   }
}
```

  - Second argument is array of pointers

- Output of example program

```
unix> cc -o foo foo.cc
unix> foo Hello World
arg #0 = foo
arg #1 = Hello
arg #2 = World
```

Functions – default arguments

- Often algorithms have optional parameters with default values
  - How to deal with these in your programs?
- Simple: in C++ functions, arguments can have default values

```
void f(double x = 5.0) ;
void g(double x, double y=3.0) ;
const int defval=3 ;
void h(int i=defval) ;

main() {
  double x(0.) ;

  f() ;       // calls f(5.0) ;
  g(x) ;      // calls g(x,3.0) ;
  g(x,5.0) ; // calls g(x,5.0) ;
  h() ;       // calls h(3) ;
}
```

- Rules for arguments with default values
  - Default values can be literals, constants, `enum`erations or `static`s
  - Positional rule: all arguments without default values must appear to the left of all arguments with default values

Function overloading

- Often algorithms have different implementations with the same functionality

```
int minimum3_int(int a, int b, int c) {
  return (a < b ? ( a < c ? a : c ) : ( b < c ? b : c) ) ;
}

float minimum3_float(float a, float b, float c) {
  return (a < b ? ( a < c ? a : c ) : ( b < c ? b : c) ) ;
}

int main() {
  int a=3,b=5,c=1 ;
  float x=4.5,y=1.2,z=-3 ;

  int d = minimum3_int(a,b,c) ;
  float w = minimum3_float(x,y,z) ;
}
```

  - The `minimum3` algorithm would be easier to use if both implementations had the same name and the compiler would automatically select the proper implementation with each use

Function overloading

- C++ function overloading does exactly that
  - Reimplementation of example with function overloading

```
int minimum3(int a, int b, int c) {
  return (a < b ? ( a < c ? a : c )
                : ( b < c ? b : c) ) ;
}

float minimum3 (float a, float b, float c) {
  return (a < b ? ( a < c ? a : c )
                : ( b < c ? b : c) ) ;
}

int main() {
  int a=3,b=5,c=1 ;
  float x=4.5,y=1.2,z=-3 ;

  int d = minimum3(a,b,c) ;
  float w = minimum3(x,y,z) ;
}
```

Overloaded functions have same name, but different signature (list of arguments)

Code calls same function name twice. Compiler selects appropriate overloaded function based on argument list

Function overloading resolution

- How does the compiler match a list of arguments to an implementation of an overloaded function? It tries

Rank	Method	Example
1	Exact Match	
2	Trivial argument conversion	`int → int&`
3	Argument Promotion	`float → double`
4	Standard argument conversion	`int→bool, double→float`
5	User defined argument conversion	(we'll cover this later)

  – Example

```
void func(int i) ;
void func(double d) ;

int main() {
  int i ;
  float f ;

  func(i) ; // Exact Match
  func(f) ; // Promotion to double
```

Function overloading – some fine points

- Functions can not be overloaded on return type

```
int function(int x) ;
float function(int x) ; // ERROR – only return type is different
```

- A call to an overloaded function is only legal if there is exactly one way to match that call to an implementation

```
void func(bool i) ;
void func(double d) ;

int main() {
  bool b ;
  int i ;
  float f ;

  func(b) ;  // Exact match
  func(f) ; // Unique Promotion to double
  func(i) ;  // Ambiguous Std Conversion (int→bool or int→double)
```

  – Its gets more complicated if you have >1 arguments

Pointers to functions

- You can create pointer to functions too!
  - Declaration

```
Type (*pfname)(Type arg1, Type arg2,…) ;
```

  - Example

```
double square(double x) {
   return x*x ;
}

int main() {
   double (*funcptr)(double i) ;      // funcptr is function ptr
   funcptr = &square ;

   cout << square(5.0) << endl ;      // Direct call
   cout << (*funcptr)(5.0) << endl ; // Call via pointer
}
```

  - Allows to pass function as function argument, e.g. to be used as callback function

Pointers to functions – example use

- Example of pointer to function – call back function

```
void sqrtArray(double x[], int len, void (*handler)(double x)) {
  int i ;
  for (i=0 ; i<len ; i++) {
    if (x[i]<0) {
      handler(x[i]) ; // call handler function if x<0
    } else {
      cout << "sqrt(" << x[i] << ") = " << sqrt(x[i]) << endl ;
    }
  }
}

void errorHandler(double x) {
  cout << "something is wrong with input value " << x << endl ;
}

int main() {
  double x[5] = { 0, 1, 2, -3, -4 } ;
  sqrtArray(x , 5,  &errorHandler) ;
}
```

Organizing your code into modules

- For all but the most trivial programs it is not convenient to keep all C++ source code in a single file
  - Split source code into multiple files

- Module: unit of source code offered to the compiler
  - Usually module = file

- How to split your code into files and modules
  1. Group functions with related functionality into a single file
     - Follow guide line 'strong cohesion', 'loose coupling'
     - Example: a collection of char* string manipulation functions go together in a single module
  2. Separate declaration and definition in separate files
     - Declaration part to be used by other modules that interact with given module
     - Definition part only offered once to compiler for compilation

Typical layout of a module

- Declarations file

```
// capitalize.hh
void convertUpper(char* str) ;
void convertLower(char* str) ;
```

Declarations

- Definitions file

```
// capitalize.cc
#include "capitalize.hh"
void convertUpper(char* ptr) {
   while(*ptr) {
      if (*ptr>='a'&&*ptr<='z') *ptr -= 'a'-'A' ;
      ptr++ ;
   }
}
void convertLower(char* ptr) {
   while(*ptr) {
      if (*ptr>='A'&&*ptr<='Z') *ptr += 'a'-'A' ;
      ptr++ ;
   }
}
```

Definitions

Using the preprocessor to include declarations

- The C++ preprocessor `#include` directive can be used to include declarations from an external module

```
// demo.cc

#include "capitalize.hh"

int main(int argc, const char* argv[]) {
    if (argc!=2) return 0 ;
    convertUpper(argv[1]) ;
    cout << argv[1] << endl ;
}
```

- But watch out for multiple inclusion of same source file
  - Multiple inclusion can have unwanted effects or lead to errors
  - Preferred solution: add safeguard in `.hh` file that gracefully handles multiple inclusions
    - rather than rely on cumbersome bookkeeping by module programming

Safeguarding against multiple inclusion

- Automatic safeguard against multiple inclusion
  - Use preprocessor conditional inclusion feature

```
#ifdef NAME
(#else)
#endif
```

  - NAME can be defined with `#define`

- Application in `capitalize.hh` example
  - If already included, CAPITALIZE_HH is set and future inclusion will be blank

```
// capitalize.hh
#ifndef CAPITALIZE_HH
#define CAPITALIZE_HH

void convertUpper(char* str) ;
void convertLower(char* str) ;

#endif
```

Namespaces

- Single global namespace often bad idea
  - Possibility for conflict: someone else (or even you inadvertently) may have used the name want you use in your new piece of code elsewhere → Linking and runtime errors may result
  - Solution: make separate 'namespaces' for unrelated modules of code
- The namespace feature in C++ allows you to explicitly control the scope of your symbols
  - Syntax:

```
namespace name {

int global = 0 ;

void func() {
   // code
   cout << global << endl ;
}


}
```

Code can access symbols inside same namespace without further qualifications

Namespaces

- But code outside namespace must explicitly use scope operator with namespace name to resolve symbol

```
namespace foo {

  int global = 0 ;

  void func() {
     // code
     cout << global << endl ;
  }

}

void bar() {
   cout << foo::global << endl ;

   foo::func() ;
}
```

← Namespace applies to functions too!

Namespace rules

- Namespace declaration must occur at the global level

```
void function foo() {
   namespace bar {     ERROR!
      statements
   }
}
```

- Namespaces are extensible

```
namespace foo {
  int bar = 0 ;
}

// other code

namespace foo {  Legal
  int foobar = 0 ;
}
```

Namespace rules

- Namespaces can nest

```
namespace foo {
  int zap = 0 ;

   namespace bar {
     int foobar = 0 ;
   }

}

int main() {
   cout << foo::zap << endl ;
   cout << foo::bar::foobar << endl ;
}
```

Legal

Recursively use :: operator to resolve nested namespaces

Namespace rules

- Namespaces can be unnamed!
  - Primary purpose: to avoid 'leakage' of private global symbols from module of code

```
namespace {
  int bar = 0 ;
}

void func() {
  cout << bar << endl ;
}
```

Code in same module ***outside*** unnamed namespace can access symbols ***inside*** unnamed namespace

Namespaces and the Standard Library

- All symbols in the Standard library are wrapped in the namespace ‘std’
- The ‘Hello world’ program revisited:

```
// my first program in C++
#include <iostream>

int main () {
  std::cout << "Hello World!“ << std::endl;
  return 0;
}
```

Using namespaces conveniently

- It is possible to import symbols from a given namespace into the current scope
  - To avoid excessive typing and confusing due to repeated lengthy notation

```
// my first program in C++
#include <iostream>
using std::cout ;
using std::endl ;

int main () {
  cout << "Hello World!“ << endl;
  return 0;
}
```

Import selected symbols into global namespace

Imported symbols can now be used without qualification in this module

  - Can also import symbols in a local scope. In that case import valid only inside local scope

Using namespaces conveniently

- You can also import the symbol contents of an entire namespace

```
// my first program in C++
#include <iostream>
using namespace std ;

int main () {
  cout << "Hello World!“ << endl;
  return 0;
}
```

- Style tip: If possible only import symbols you need

Modules and namespaces

- Namespace enhance encapsulation of modules
  - Improved capitalize module

```
// capitalize.hh
#ifndef CAPITALIZE_HH
#define CAPITALIZE_HH
namespace capitalize {
void convertUpper(char* str) ;
void convertLower(char* str) ;
}
#endif
```

```
// capitalize.cc
#include “capitalize.hh”
namespace capitalize {
  void convertUpper(char* ptr) {
     while(*ptr) {
        if (*ptr>=‘a’&&*ptr<=‘z’) *ptr -= ‘a’-’A’ ;
        ptr++ ;
     }
  }
  void convertLower(char* ptr) {
     while(*ptr) {
        if (*ptr>=‘A’&&*ptr<=‘Z’) *ptr += ‘a’-’A’ ;
        ptr++ ;
     }
  }
}
```

The standard library as example

- Each C++ compiler comes with a standard suite of libraries that provide additional functionality
  - `<math>` -- Math routines `sin(),cos(),exp(),pow(),…`
  - `<stdlib>` -- Standard utilities `strlen(),strcat(),…`
  - `<stdio>` -- File manipulation utilities `open(),write(),close(),…`

- Nice example of modularity and use of namespaces
  - All Standard Library routines are contained in `namespace std`

Compiling & linking code in multiple modules

- Compiling & linking code in a single module
  - `g++ -c demo.cc`
    - Converts demo.cc C++ code into demo.o (machine code)
  - `g++ -o demo demo.o`
    - Links demo.o with Standard Library code and makes standalone executable code
  - Alternatively, '`g++ -o demo demo.cc`' does all in one step

- Compiling & linking code in multiple modules
  - `g++ -c module1.cc`
  - `g++ -c module2.cc`
  - `g++ -c module3.cc`
  - `g++ -o demo module1.o module2.o module3.o`
    - Link module1,2,3 to each other and the Standard Library code

Intermezzo – Concept of 'object libraries'

- Operating systems usually also support concept 'object libraries'
  - A mechanism to store multiple compile modules (.o files) into a single library file
  - Simplifies working with very large number of modules
  - Example: all Standard Library modules are often grouped together into a single 'object library file'

- Creating an object library file from object modules
  - Unix: '`ar q libLibrary.a module1.o module2.o ….`'
- Linking with a library of object modules
  - Unix: '`g++ -o demo demo.cc –lLibrary`'
    - Above syntax looks for library name `libLibrary.a` in current directory or in 'standard locations'
    - To add directory to library search path, specify –L<path> in g++ command line

3 Class Basics

Overview of this section

- Contents of this chapter
  - **structs and classes** - Grouping data and functions together
  - **public vs private** – Improving encapsulation through hiding of internal details
  - **constructors and destructors** – Improving encapsulation through self-initialization and self-cleanup
  - **more on const** – Improving modularity and encapsulation through const declarations

Encapsulation

- OO languages like C++ enable you to create your own data types. This is important because
  - New data types make program easier to visualize and implement new designs
  - User-defined data types are reusable
  - You may modify and enhance new data types as programs evolve and specifications change
  - New data types let you create objects with simple declarations

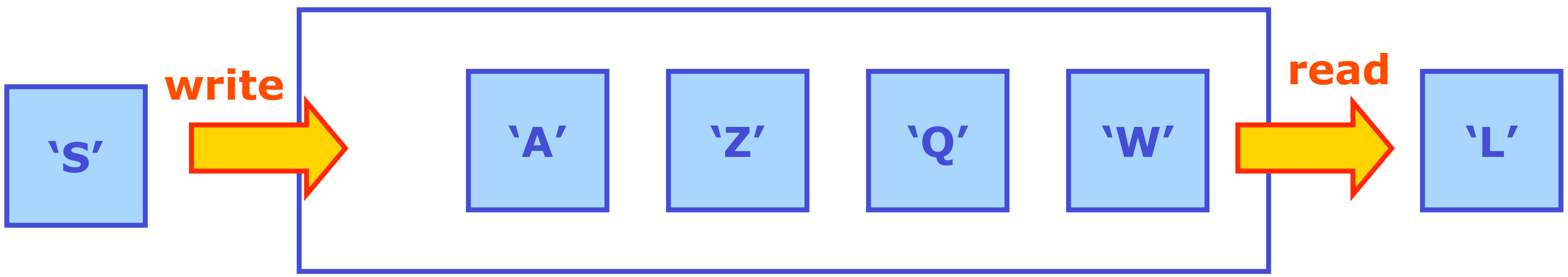
- Example

```
Window w ;      // Window object
Database ood ; // Database object
Device d ;      // Device object
```

Evolving code design through use of C++ classes

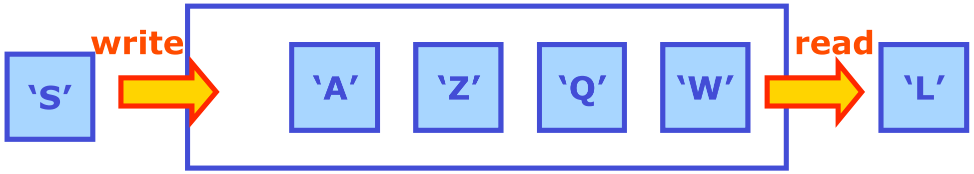
- Illustration of utility of C++ classes – Designing and building a FIFO queue
  - FIFO = '**F**irst **I**n **F**irst **O**ut'

- Graphical illustration of a FIFO queue



Evolving code design through use of C++ classes

- First step in design is to write down the *interface*
  - How will 'external' code interact with our FIFO code?



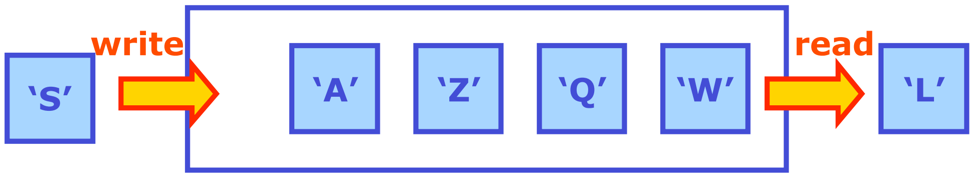
- List the essential interface tasks
  1. **Create** and initialize a FIFO
  2. **Write** a character in a FIFO
  3. **Read** a character from a FIFO
  - Support tasks
    1. How many characters are currently in the FIFO
    2. Is a FIFO empty
    3. Is a FIFO full

Designing the C++ class FIFO – interface

- List of interface tasks
  1. **Create** and initialize a FIFO
  2. **Write** a character in a FIFO
  3. **Read** a character from a FIFO

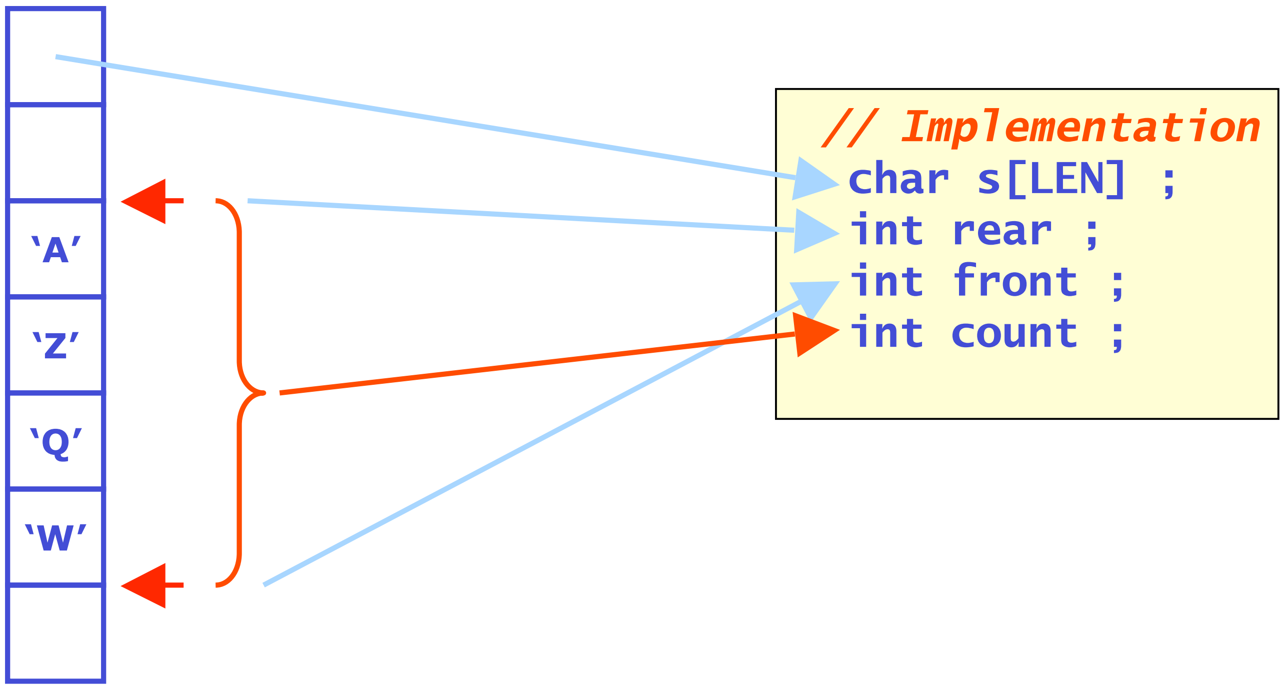
- List desired support tasks
  1. How many characters are currently in the FIFO
  2. Is a FIFO empty
  3. Is a FIFO full

```
// Interface
void init() ;
void write(char c) ;
char read() ;

int nitems() ;
bool full() ;
bool empty() ;
```



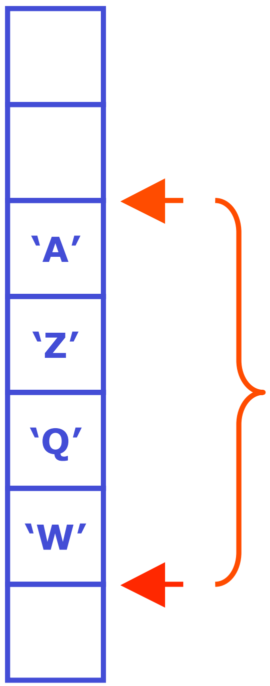
Designing the C++ struct FIFO – implementation

- Implement FIFO with array of elements
  - Use index integers to keep track of front and rear, size of queue



Designing the C++ struct FIFO – implementation

- Implement FIFO with array of elements
  - Use index integers to keep track of front and rear, size of queue
  - Indices revolve: if they reach end of array, they go back to 0

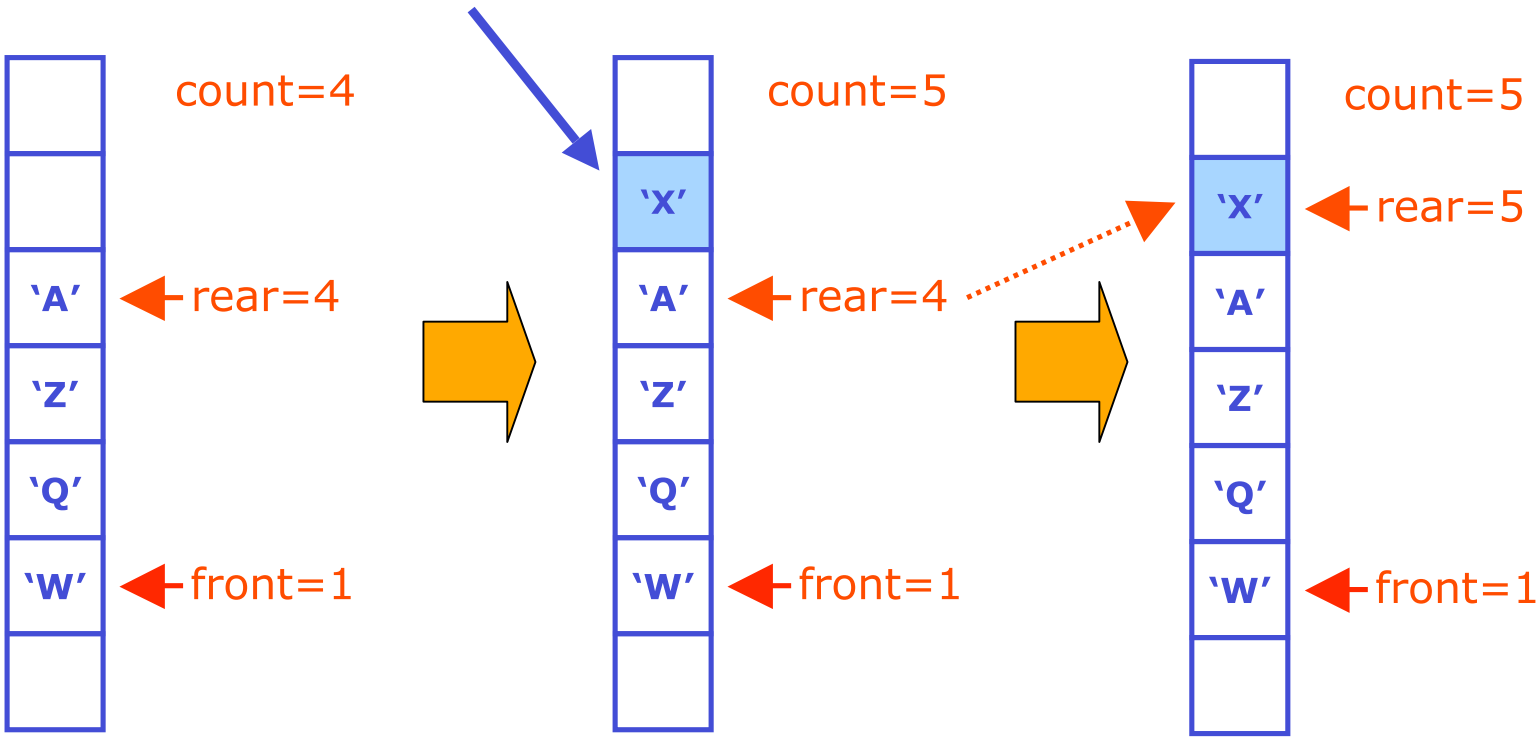



```
// Implementation
void init() { front = rear = count = 0 ; }

void write(char c) { count++ ;
                     if(rear==LEN) rear=0 ;
                     s[rear++] = c ; }

char read() { count-- ;
              if (front==LEN) front=0 ;
              return s[front++] ; }


int nitems() { return count ; }
bool full() { return (count==LEN) ; }
bool empty() { return (count==0) ; }
```

Designing the C++ struct FIFO – implementation

- Animation of FIFO write operation

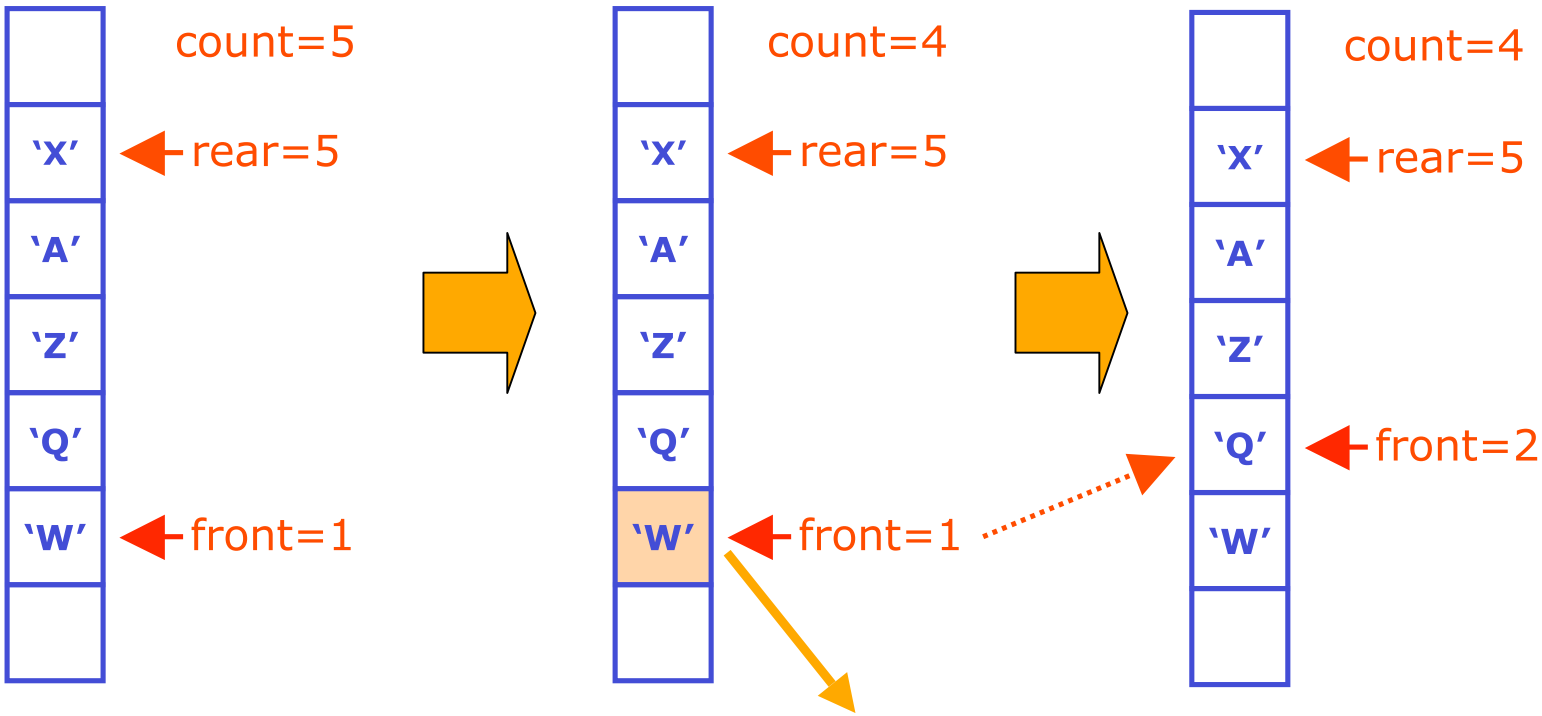
```
void write(char c) { count++ ;
                     if(rear==LEN) rear=0 ;
                     s[rear++] = c ; }
```



Designing the C++ struct FIFO – implementation

- Animation of FIFO read operation

```
char read() { count-- ;
              if (front==LEN) front=0 ;
              return s[front++] ; }
```



Putting the FIFO together – the struct concept

- The finishing touch: putting it all together in a **struct**

```
const int LEN = 80 ; // default fifo length

struct Fifo {
  // Implementation
  char s[LEN] ;
  int front ;
  int rear ;
  int count ;

  // Interface
  void init() { front = rear = count = 0 ; }
  int nitems() { return count ; }
  bool full() { return (count==LEN) ; }
  bool empty() { return (count==0) ; }
  void write(char c) { count++ ;
                       if(rear==LEN) rear=0 ;
                       s[rear++] = c ; }
  char read() { count-- ;
                if (front==LEN) front=0 ;
                return s[front++] ; }
} ;
```

Characteristics of the ‘struct’ construct

- Grouping of data members facilitates storage allocation
  - Single statement allocates all data members

```
// Allocate struct data type ‘Fifo’
Fifo f ;

// Access function through name ‘f’
f.init() ;

// Access data member through name ‘f’
cout << f.count << endl ;
```

- A `struct` organizes access to data members and functions through a common symbolic name

Characteristics of the ‘struct’ construct

- Concept of ‘member functions’ automatically ties manipulator functions to their data
  - No need to pass data member operated on to interface function

```
// Solution without
// member functions

struct fifo {
   int front, rear, count ;
} ;

char read_fifo(fifo& f) {
  f.count-- ;
  …
}

fifo f ;
read_fifo(f) ;
```

```
// Solution with
// member functions

struct fifo {
   int front, rear, count ;
   char read() {
     count-- ;
     …
   }
} ;

fifo f ;
f.read() ;
```

Using the FIFO example code

- Example code using the FIFO struct

```
const char* data = “data bytes” ;
int i, nc = strlen(data) ;

Fifo f ;
f.init() ; // initialize FIFO

// Write chars into fifo
const char* p = data ;
for (i=0 ; i<nc && !f.full() ; i++) {
  f.write(*p++) ;
}

// Count chars in fifo
cout << f.nitems() << “ characters in fifo” << endl ;

// Read chars back from fifo
for (i=0 ; i<nc && !f.empty() ; i++) {
  cout << f.read() << endl ;
}
```

Program Output

```
10 characters in fifo
data bytes
```

Characteristics of the FIFO code

- Grouping data, function members into a struct promotes encapsulation
  - All data members needed for `fifo` operation allocated in a single statement
  - All data objects, functions needed for `fifo` operation have implementation contained within the namespace of the FIFO object
  - Interface functions associated with `struct` allow implementation of a controlled interface functionality of FIFO
    - For example can check in read(), write() if FIFO is full or empty and take appropriate action depending on status

- Problems with current implementation
  - User needs to explicitly initialize `fifo` prior to use
  - User needs to check explicitly if `fifo` is not full/empty when writing/reading
  - Data objects used in implementation are visible to user and subject to external modification/corruption

Controlled interface

- Improving encapsulation
  - We improve encapsulation of the FIFO implementation by restricting access to the member functions and data members that are needed for the implementation
- Objective – **a controlled interface**
  - With a controlled interface, i.e. designated member functions that perform operations on the FIFO, we can catch error conditions on the fly and validate offered input before processing it
  - With a controlled interface there is no 'back door' to the data members that implement the `fifo` thus guaranteeing that no corruption through external sources can take place
    - NB: This also improves performance you can afford to be less paranoid.

Private and public

- C++ access control keyword: ‘**public**’ and ‘**private**’

```
struct Name {
private:

… members … // Implementation

public:

… members … // Interface

} ;
```

- Public data
  - Access is unrestricted. Situation identical to no access control declaration
- Private data
  - Data objects and member functions in the private section can only accessed by member functions of the struct (which themselves can be either private or public)

Redesign of fifo class with access restrictions

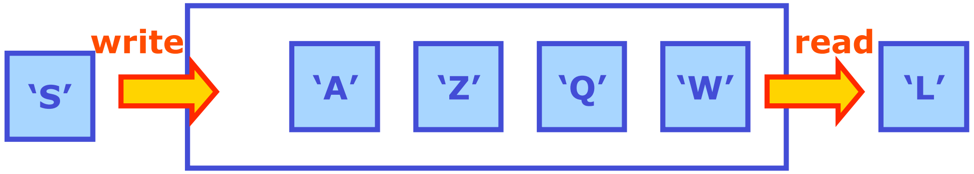
```
const int LEN = 80 ; // default fifo length

struct Fifo {
  private:   // Implementation
  char s[LEN] ;
  int front ;
  int rear ;
  int count ;

  public:    // Interface
  void init() { front = rear = count = 0 ; }
  int nitems() { return count ; }
  bool full() { return (count==LEN) ; }
  bool empty() { return (count==0) ; }
  void write(char c) { count++ ;
                       if(rear==LEN) rear=0 ;
                       s[rear++] = c ; }
  char read() { count-- ;
                if (front==LEN) front=0 ;
                return s[front++] ; }
} ;
```

Using the redesigned FIFO struct

- Effects of access control in improved fifo struct

```
Fifo f ;
f.init() ;                      // initialize FIFO


f.front = 5 ;                   // COMPILER ERROR – not allowed
cout << f.count << endl ;       // COMPILER ERROR – not allowed

cout << f.nitems() << endl ; // OK – through
                             // designated interface
```

`front` is an implementation details that's not part of the abstract FIFO concept. Hiding this detail promotes encapsulation as we are now able to change the implementation later with the certainty that we will not break existing code



Class – a better struct

- In addition to 'struct' C++ also defines 'class' as a method to group data and functions
  - In structs members are by default public,
    In classes member functions are by default private
  - Classes have several additional features that we'll cover shortly

Equivalent

```
struct Name {
private:

… members …

public:

… members …
} ;
```

```
class Name {

… members …

public:

… members …

} ;
```

Classes and namespaces

- Classes (and structs) also define their own `namespace`
  - Allows to separate interface and implementation even further by separating declaration and definition of member functions

Declaration and definition

```
class Fifo {
public:    // Interface
char read() {
  count-- ;
  if (front==len) front=0 ;
  return s[front++] ;
  }
} ;
```

Declaration only

```
class Fifo {
public:    // Interface
char read() ;
} ;
```

Definition

```
#include "fifo.hh"
char Fifo::read() {
  count-- ;
  if (front==len) front=0 ;
  return s[front++] ;
  }
```

Use of scope operator :: to specify read() function of Fifo class when outside class definition

Classes and namespaces

- Scope resolution operator can also be used in class member function to resolve ambiguities

```
class Fifo {
public:    // Interface
char read() {
  …
  std::read() ;
  …
  }
} ;
```

Use scope operator to specify that you want to call the read() function in the std namespace rather than yourself

Classes and files

- Class declarations and definitions have a natural separation into separate files
  - A header file with the class declaration
    To be included by everybody that uses the class
  - A definition file with definition that is only offered once to the compiler
  - Advantage: You do not need to recompile code using class `fifo` if only implementation (file `fifo.cc`) changes

fifo.hh

```
#ifndef FIFO_HH
#define FIFO_HH
class Fifo {
public:    // Interface
char read() ;
} ;
#endif
```

fifo.cc

```
#include "fifo.hh"
char Fifo::read() {
  count-- ;
  if (front==len) front=0 ;
  return s[front++] ;
  }
```

Constructors

- Abstraction of FIFO data typed can be further enhanced by letting it take care of its own initialization
  – User should not need to know if and how initialization should occur
  – Self-initialization makes objects easier to use and gives less chances for user mistakes

- C++ approach to self-initialization – the Constructor member function
  – Syntax: member function with function name identical to class name

```
class ClassName {
…
ClassName() ;
…
} ;
```

Adding a Constructor to the FIFO example

- Improved FIFO example

```
class Fifo {
public:
  void init() ;
  …
```

⇨

```
class Fifo {
public:
  Fifo() { init() ; }

private:
  void init() ;
  …
```

- Simplified use of FIFO

```
Fifo f ;   // creates raw FIFO
f.init() ; // initialize FIFO
```

⇩

```
Fifo f ;   // creates initialized FIFO
```

Default constructors vs general constructors

- The FIFO code is an example of a **default constructor**
  - A default constructor by definition takes no arguments

- Sometimes an objects requires user input to properly initialize itself
  - Example: A class that represents an open file – Needs file name
  - Use ‘regular constructor’ syntax

```
class ClassName {
…
ClassName(argument1,argument2,…argumentN) ;
…
} ;
```

  - Supply constructor arguments at construction

```
ClassName obj(arg1,…,argN) ;
ClassName* ptr = new ClassName(Arg1,…ArgN) ;
```

Constructor example – a File class

```
class File {

private:
   int fh ;

public:
   File(const char* name) {
      fh = open(name) ;
   }

   void read(char* p, int n) { ::read(fh,p,n) ; }
   void write(char* p, int n) { ::write(fh,p,n) ; }
   void close() { ::close(fh) ; }
} ;
```

```
File* f1 = new File("dbase") ;
File f2("records") ;
```

Supply constructor arguments here

Multiple constructors

- You can define multiple constructors with different signatures
  - C++ function overloading concept applies to class member functions as well, including the constructor function

```
class File {

private:
   Int fh ;

public:
   File() {
      fh = open("Default.txt") ;
   }
   File(const char* name) {
      fh = open(name) ;
   }

   read(char* p, int n) { ::read(p,n) ; }
   write(char* p, int n) { ::write(p,n) ; }
   close() { ::close(fh) ; }
} ;
```

Default constructor and default arguments

- Default values for function arguments can be applied to all class member functions, including the constructor
  - If any constructor can be invoked with no arguments (i.e. it has default values for all arguments) it is also the default constructor

```
class File {

private:
   Int fh ;

public:
File(const char* name=“Default.txt”) {
      fh = open(name) ;
   }

   read(char* p, int n) { ::read(p,n) ; }
   write(char* p, int n) { ::write(p,n) ; }
   close() { ::close(fh) ; }
} ;
```

Default constructors and arrays

- Array allocation of objects does not allow for specification of constructor arguments

```
Fifo* fifoArray = new Fifo[100] ;
```

- **You can only define arrays of classes that have a default constructor**
  - Be sure to define one if it is logically allowed
  - Workaround for arrays of objects that need constructor arguments: allocate array of pointers ;

```
Fifo** fifoPtrArray = new (Fifo*)[100] ;
int i ;
for (i=0 ; i<100 ; i++) {
   fifoPtrArray[i] = new Fifo(arguments…) ;
}
```

  - Don't forget to `delete` elements in addition to array afterwards!

Classes contained in classes – member initialization

- If classes have other classes w/o default constructor as data member you need to initialize 'inner class' in constructor of 'outer class'

```
class File {
  public:
  File(const char* name) ;
  …
} ;

class Database {
  public:
  Database(const char* fileName) ;

  private:
  File f ;
} ;

Database::Database(const char* fileName) : f(fileName) {
  // Database constructor
}
```

Class member initialization

- General constructor syntax with member initialization

```
ClassName::ClassName(args) :
   member1(args),
   member2(args), …
   memberN(args) {
   // constructor body
}
```

  - Note that insofar order matters, data members are initialized in ***the order they are declared in the class***, not in the order they are listed in the initialization list in the constructor
  - Also for basic types (and any class with default ctor) the member initialization form can be used

Initialization through assignment

```
File(const char* name) {
      fh = open(name) ;
   }
```

Initialization through constructor

```
File(const char* name) :
fh(open(name)) {
}
```

  - Performance tip: for classes constructor initialization tends to be faster than assignment initialization (more on this later)

Common initialization in multiple constructors

- Overlapping functionality is a common design issue with multiple constructors
  - How to avoid unnecessary code duplication (i.e member initialization)
- Common mistake – attempts to make one constructor function call another one

```
class Array {
public:
  Array(int size) {
    _size = size ;
    _x = new double[size] ;
  }

  Array(const double* input, int size) : Array(size) {
    int i ;
    for (i=0 ; i<size ; i++) _x[i] = input[i] ;
  }

private
  int _size ;
  double* _x ;
}
```

Not Allowed!!! (Compiler Error)

Common initialization in multiple constructors

- Another clever but wrong solution
  - Idea: Call `Array(size)` as if it were a regular member function, which will then perform the necessary initialization steps

```
Array(const double* input, int size) {

  Array(size) ; // This doesn't work either!

  int i ;
  for (i=0 ; i<size ; i++) _x[i] = input[i] ;
}
```

  - Problem: It is legal C++ (it compiles fine) but it *doesn't do what you think it does*!
  - Calling a constructor like this creates a temporary object that is initialized with size and immediately destroyed again. It does not initialize the instance of array you are constructing with the `Array(double*,int)` constructor

Common initialization in multiple constructors

- The correct solution is to make a private initializer function that is called from all relevant constructors

```
class Array {
public:
  Array(int size) {
    initialize(size) ;
  }

  Array(const double* input, int size) {
    initialize(size) ;
    int i ;
    for (i=0 ; i<size ; i++) _x[i] = input[i] ;
  }

private:
  void initialize(int size) {
    _size = size ;
    _x = new double[size] ;
  }
  int _size ;
  double* _x ;
}
```

Destructors

- Classes that define constructors often allocate dynamic memory or acquire resources
  - Example: File class acquires open file handles, any other class that allocates dynamic memory as working space

- C++ defines Destructor function for each class to be called at end of lifetime of object
  - Can be used to release memory, resources before death

- Class destructor syntax:

```
class ClassName {
…
~ClassName() ;
…
} ;
```

Example of destructor in File class

```
class File {

private:
 int fh ;
 close() { ::close(fh) ; }

public:
   File(const char* name) { fh = open(name) ; }
   ~File() { close() ; }
   …
} ;
```

File is automatically closed when object is deleted

```
void readFromFile() {
   File *f = new File("theFile.txt") ;
   // read something from file
   delete f ;
}
```

Opens file automatically

Closes file automatically

Automatic resource control

- Destructor calls can take care of automatic resource control
  - Example with dynamically allocated `File` object

```
void readFromFile() {
   File *f = new File("theFile.txt") ;
   // read something from file
   delete f ;
}
```

Opens file automatically

Closes file automatically

  - Example with automatic `File` object

```
void readFromFile() {
   File f("theFile.txt") ;
   // read something from file
}
```

Opens file automatically

Deletion of automatic variable f calls destructor & closes file automatically

  - Great example of abstraction of file concept and of encapsulation of resource control

Classes vs *Instances* – an important concept

- There is an important distinction between *classes* and *instances* of classes (objects)
  - A class is a unit of code
  - An instance is an object in memory that is managed by the class code

```
Array a ;    // creates an array object
```

Class   Instance

- A class can have more than one instance

```
Array a1 ;   // first instance
Array a2 ;   // second instance
```

Classes vs *Instances* – an important concept

- The concept that a single unit of code can work with multiple objects in memory has profound consequences
  - Start with program that makes two arrays like this

```
Array a1 ;   // first instance
Array a2 ;   // second instance
```

  - Now what happens inside the array's `initialize()` code

```
void Array::initialize(int size) {
   _size = size ;
   _x = new double[size] ;
 }
```

  - *Q: To which memory object does data member `_size` belong, **`a1`** or **`a2`**?*
  - *A: **It depends on who calls `initialize()`!***

    If you call `a1.initialize()` data member `_size` automatically refers to `a1._size`, if you call `a2.initialize()` it refers to `a2._size` etc…
  - Concept is called 'automatic binding'

Intermezzo – Referring to yourself – this

- Can you figure which instance you are representing in a member function? A: Yes, using the special object `this`
  - The '**`this`**' keyword return a pointer to yourself inside a member function

```
void Array::initialize() {
   cout << "I am a array object, my pointer is " << this << endl ;
}
```

- How does it work?
  - In case you called `a1.initialize()` from the main program, `this=&a1`
  - In case
    you called `a2.initialize()` then `this=&a2` etc…

Intermezzo – Referring to yourself – this

- You don't need *this* very often.
  - If you think you do, think hard if you can avoid it, you usually can
- Most common cases where you really need *this* are
  - Identifying yourself to an outside function (see below)
  - In assignment operations, to check that you're not copying onto yourself (e.g. `a1=a1`). We'll come back to this later
- How to identify yourself to the outside world?
  - Example: Member function of `classA` needs to call external function `externalFunc()` that takes reference to `classA`

```
void externalFunction(ClassA& obj) {
  …
}

void classA::memberFunc() {
  if (certain_condition) {
    externFunction(*this) ;
  }
}
```

Copy constructor – a special constructor

- The copy constructor is the constructor with the signature

```
ClassA::ClassA(const ClassA&) ;
```

- It is used to make a clone of your object

```
ClassA a ;
ClassA aclone(a) ; // aclone is an identical copy of a
```

- It exists for all objects because the C++ compiler provides a *default implementation* if you don't supply one
  - The default copy constructor calls the copy constructor for all data members. Basic type data members are simply copied
  - The default implementation is not always right for your class, we'll return to this shortly

Taking good care of your property

- Use 'ownership' semantics in classes as well
  - Keep track of who is responsible for resources allocated by your object
  - The constructor and destructor of a class allow you to automatically manage your initialization/cleanup
  - All private resources are always owned by the class so make sure that the destructor always releases those

- Be careful what happens to 'owned' objects when you make a copy of an object
  - Remember: default copy constructor calls copy ctor on all class data member and copies values of all basic types
  - **Pointers are basic types**
  - If an 'owned' pointer is copied by the copy constructor it is no longer clear which instance owns the object → **danger ahead!**

Taking good care of your property

- Example of default copy constructor wreaking havoc

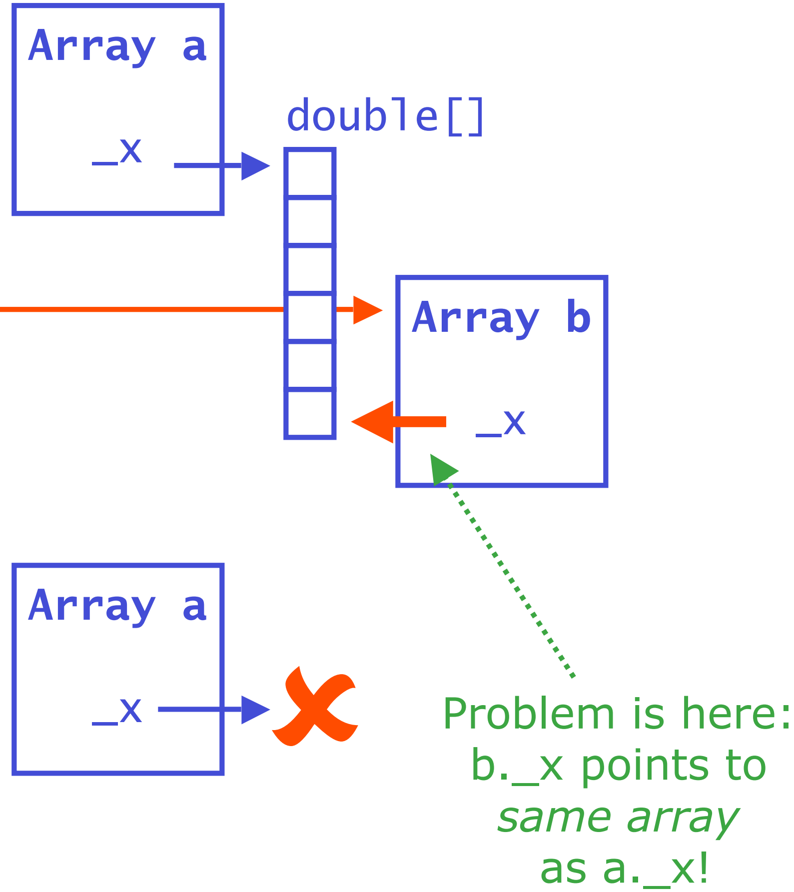
```
class Array {
public:
  Array(int size) {
    initialize(size) ;
  }
  ~Array() {
    delete[] _x ;
  }

private:
  void initialize(int size) {
    _size = size ;
    _x = new double[size] ;
  }
  int _size ;
  double* _x ;
}
```

Watch out! Pointer data member

Taking good care of your property

- Add illustration

```
void example {

Array a(10) ;
// 'a' Constructor allocates _x ;

if (some_condition)
  Array b(a) ;
  // 'b' Copy Constructor does
  // b._x = a._x ;

  // b appears to be copy of a
}
// 'b' Destructor does
// delete[] _b.x

// BUT _b.x == _a.x → Memory
// allocated by 'Array a' has
// been released by ~b() ;

<Do something with Array>
// You are dead!
}
```



Taking good care of your property

- Example of default copy constructor wreaking havoc

```
class Array {
public:
  Array(int size) {
    initialize(size) ;
  }
  ~Array
    dele
  }

private:
  void i
    _siz
    _x =
  }
  int _s
  double* _x ;
}
```

```
void example {

Array a(10) ;
// 'a' Constructor allocates _x ;
```

Whenever your class owns dynamically allocated memory or similar resources you need to implement your own copy constructor!

```
// BUT _b.x == _a.x → Memory
// allocated by 'Array a' has
// been released by ~b() ;

<Do something with Array>
// You are dead!
}
```

Example of a custom copy constructor

```
class Array {
public:
  Array(int size) {
    initialize(size) ;
  }

  Array(const double* input, int size) {
    initialize(size) ;
    int i ;
    for (i=0 ; i<size ; i++) _x[i] = input[i] ;
  }

  Array(const Array& other) {
    initialize(other._size) ;
    int i ;
    for (i=0 ; i<_size ; i++) _x[i] = other._x[i] ;
  }

private:
  void initialize(int size) {
    _size = size ;
    _x = new double[size] ;
  }
  int _size ;
  double* _x ;
}
```

Classes vs Instances
Here we are dealing explicitly with **one** class and **two** instances

Symbol **`_x`** refers to data member of **`this`** instance

Symbol **`other._x`** refers to data member of **`other`** instance

Another solution to copy constructor problems

- You can disallow objects being copied by declaring their copy constructor as 'private'
  - Use for classes that should not copied because they own non-clonable resources or have a unique role
  - Example: class `File` – logistically and resource-wise tied to a single file so a clone of a `File` instances tied to the same file make no sense

```
class File {

private:
 Int fh ;
 close() { ::close(fh) ; }
 File(const File&) ; // disallow copying

public:
   File(const char* name) { fh = open(name) ; }
   ~File() { close() ; }
   …
} ;
```

Ownership and defensive programming

- Coding mistakes happen, but by programming defensively you will spot them easier
  - Always initialize owned pointers to zero if you do not allocate your resources immediately
  - Always set pointers to zero after you delete the object they point to
- By following these rules you ensure that you never have 'dangling pointers'
  - *Dangling pointers* = Pointers pointing to a piece memory that is no longer allocated which may return random values
  - Result – more predictable behavior
  - Dereferencing a dangling pointer may
    - Work just fine in case the already released memory has not been overwritten yet
    - Return random results
    - Cause your program to crash
  - Dereferencing a zero pointer will always terminate your program immediately in a clean and understandable way

Const and Objects

- ‘const’ is an important part of C++ interfaces.
  - It promotes better modularity by enhancing ‘loose coupling’

- Reminder: const and function arguments

```
void print(int value)          // pass-by-value, value is copied

void print(int& value)         // pass-by-reference,
                                  print may change value
void print(const int& value) // pass-by-const-reference,
                                  print may not change value
```

- Const rules simple to enforce for basic types: ‘=‘ changes contents
  - Compile can look for assignments to const reference and issue error
  - What about classes? Member functions may change contents, difficult to tell?
  - How do we know? We tell the compiler which member functions change the object!

Const member functions

- By default all member functions of an object are presumed to change an object
  - Example

```
class Fifo {
  …
  void print() ;
  …
}

main() {
  Fifo fifo ;
  showTheFifo(fifo) ;
}

void showTheFifo(const Fifo& theFifo)
{
    theFifo.print() ; // ERROR – print() is allowed
                      // to change the object
}
```

Const member functions

- Solution: declare `print()` to be a member function that does not change the object

```
class Fifo {
  …
  void print() const ;
  …
}

main() {
  Fifo fifo ;
  showTheFifo(fifo) ;
}

void showTheFifo(const Fifo& theFifo)
{
    theFifo.print() ; // OK print() does not change object
}
```

A member function is declared const by putting 'const' behind the function declaration

Const member function – the flip side

- The compiler will enforce that no statement inside a const member function modifies the object

```
class Fifo {
  …
  void print() const ;
  …
  int size ;
}

void Fifo::print() const {
   cout << size << endl ; // OK
   size = 0 ;             // ERROR const function is not
                             allows to modify data member
}
```

Const member functions – indecent exposure

- Const member functions are also enforced not to 'leak' non-const references or pointers that allows users to change its content

```
class Fifo {
 …
 char buf[80] ;
 …
 char* buffer() const {
    return buf ; // ERROR – Const function exposing
                    non-const pointer to data member
  }
}
```

Const return values

- Lesson: Const member functions can only return const references to data members
  - Fix for example of preceding page

This const says that this member function will not change the Fifo object

```
class Fifo {
  …
  char buf[80] ;
  …
  const char* buffer() const {
     return buf ; // OK
  }
}
```

This const says the returned pointer cannot be used to modify what it points to

Why const is good

- Getting all your const declarations in your class correct involves work! – Is it work the trouble?
- Yes! – Const is an important tool to promote encapsulation
  - Classes that are 'const-correct' can be passed through const references to functions and other objects and retain their full 'read-only' functionality
  - Example

```
main() {
  Fifo fifo ;
  showTheFifo(fifo) ;
}

void showTheFifo(const Fifo& theFifo)
{
    theFifo.print() ;
}
```

  - Const correctness of class `Fifo` loosens coupling between `main()` and `showTheFifo()` since `main()`'s author does not need to closely follow if future version of `showTheFifo()` may have undesirable side effects on the object

Mutable data members

- Occasionally it can be useful to be able to modify selected data members in a const object
  - Most frequent application: a cached value for a time-consuming operation
  - Your way out: declare that data member 'mutable'. In that case it can be modified even if the object itself is const

```
class FunctionCalculation {
 …
 mutable float cachedResult ;
 …
 float calculate() const {
   // do calculation
   cachedResult = <newValue> ; // OK because cachedResult
                               // is declared mutable
   return cachedResult ;
 }
}
```

  - Use sparingly!

Static data members

- OO programming minimizes use of global variables because they are problematic
  - Global variable cannot be encapsulated by nature
  - Changes in global variables can have hard to understand side effects
  - Maintenance of programs with many global variables hard
- C++ preferred alternative: static variables
  - A static data member encapsulates a variable inside a class
    - Optional 'private' declaration prevents non-class members to access variable
  - A static data member is shared by all instances of a class
  - Syntax

```
class ClassName {
  …
  static Type Name ;      // Declaration
  …
}

Type ClassName::Name = value ;      // Definition and initialization
```

Static data members

- Don’t forget definition in addition to declaration!
  - Declaration in class (in .hh) file. Definition in .cc file
- Example use case:
  - class that keeps track of number of instances that exist of it

```
class Counter {
public:
  Counter() { count++ ; }
  ~Counter() { count-- ; }

  void print() {
    cout << “there are “
         << count
         << “instances of count
         << endl ;
  }
private:
  static int count ;
} ;

int Counter::count = 0 ;
```

```
int main() {
  Counter c1 ;
  c1.Print() ;

  if (true) {
    Counter c2,c3,c4 ;
    c1.Print() ;
  }
  c1.Print() ;
  return 0 ;
}
```

```
there are 1 instances of count
there are 4 instances of count
there are 1 instances of count
```

Static function members

- Similar to static data member, static member functions can be defined
  - Syntax like regular function, with static keyword prefixed in declaration only

```
class ClassName {
  …
  static Type Name(Type arg,…) ;
  …
}

type ClassName::Name(Type arg,…) {
  // body goes here
}
```

  - Static function can ***access static data members only*** since function is not associated with particular instance of class
  - Can call function without class instance

```
ClassName::Name(arg,…) ;
```

Static member functions

- Example use case – modification of preceding example

```
class Counter {
public:
  Counter() { count++ ; }
  ~Counter() { count-- ; }
  static void print() {
    cout << “there are “
         << count
         << “instances of count
         << endl ;
  }
private:
  static int count ;
} ;

Counter::count = 0 ;
```

```
int main() {
  Counter::print() ;

  Counter c1 ;
  Counter::print() ;

  if (true) {
    Counter c2,c3,c4 ;
    Counter::print() ;
  }
  Counter::print() ;
  return 0 ;
}
```

```
there are 0 instances of count
there are 1 instances of count
there are 4 instances of count
there are 1 instances of count
```

4 Class Analysis & Design

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

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?

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

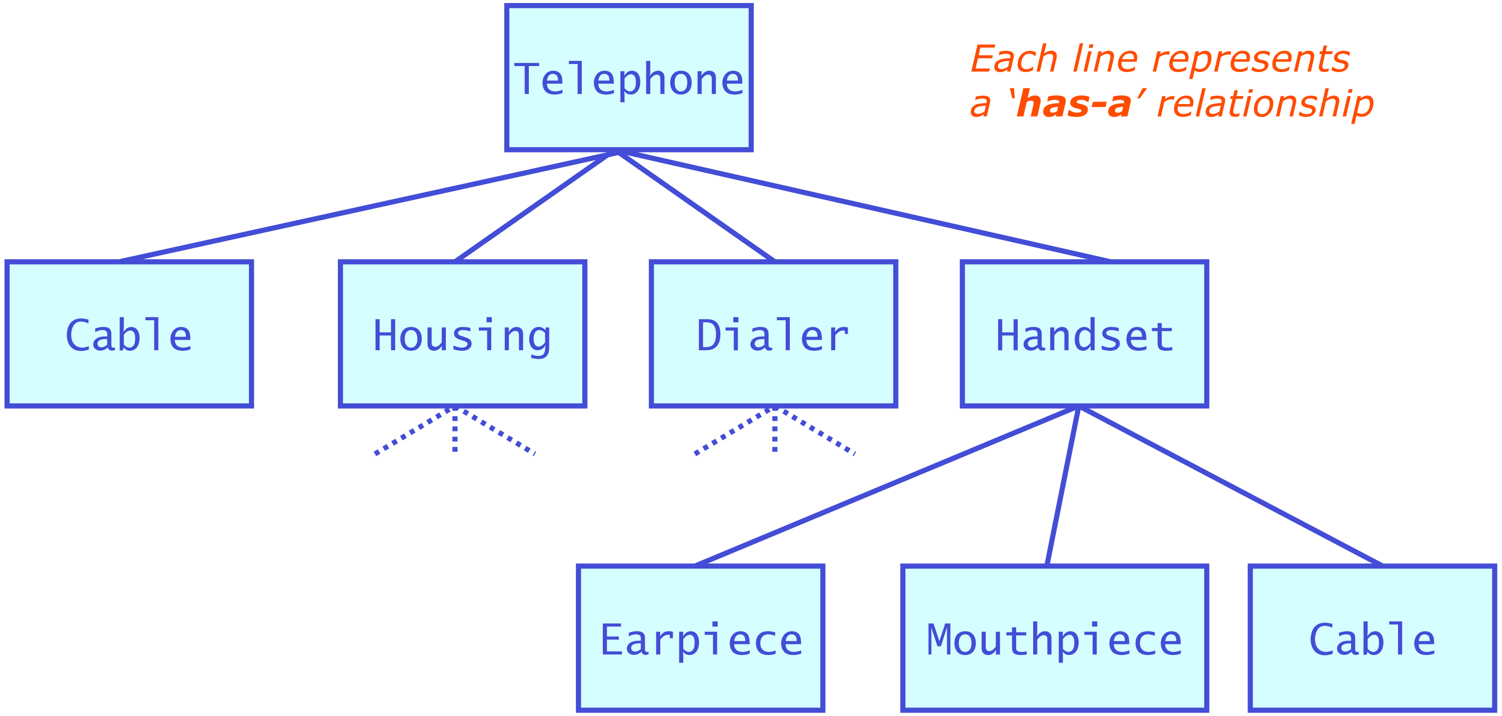
2. Associate software objects with natural objects ('objects in the application domain')
   - Actions translate to member functions
   - Attributes translate to data members

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

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

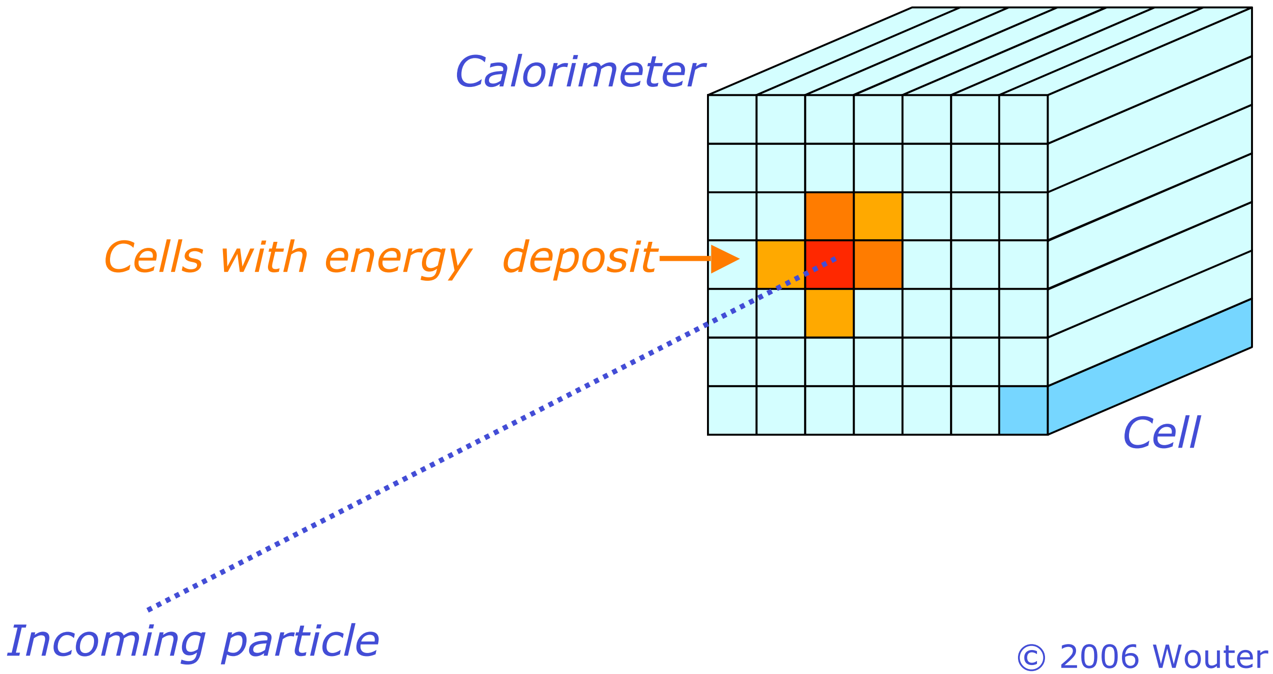


*Each line represents a ‘**has-a**’ relationship*

Analysis – Example from High Energy Physics

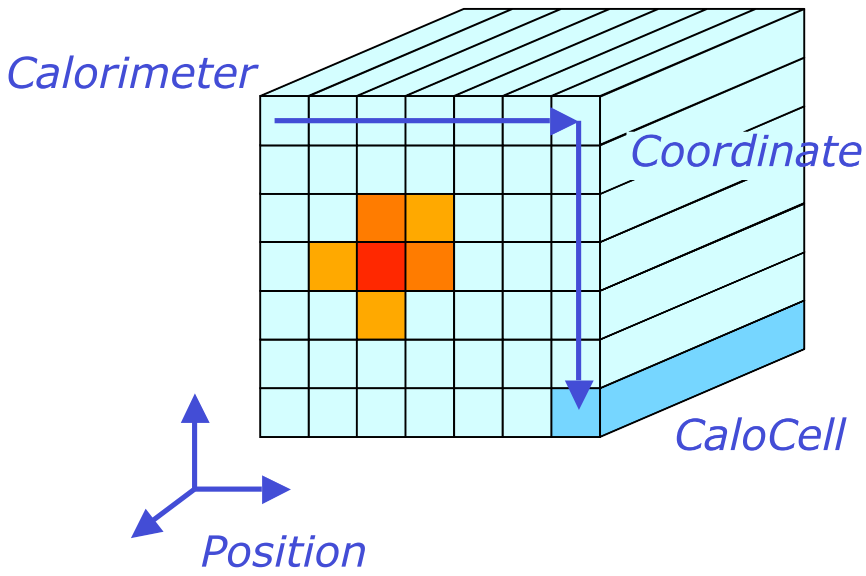
- Real life often not so clean cut

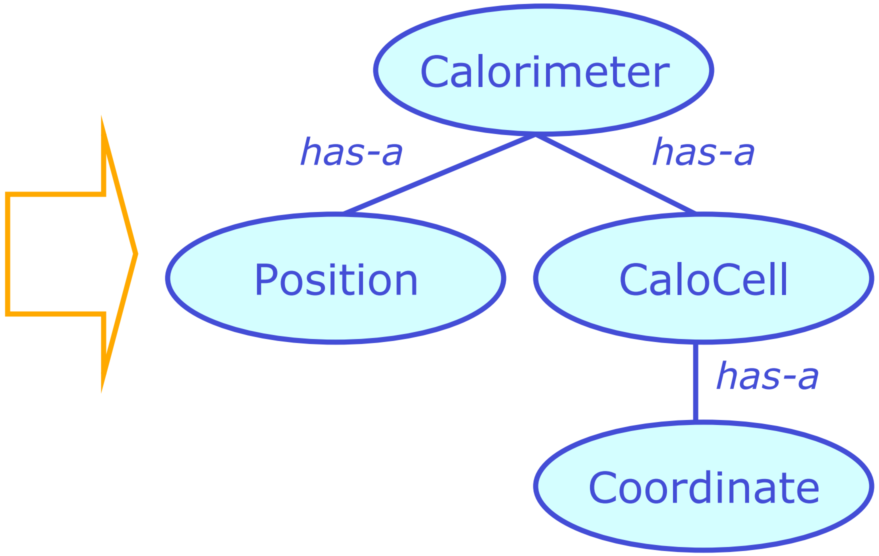
- Example problem from High Energy physics
  - We have a file with experimental data from a calorimeter.
  - A calorimeter is a HEP detector that detects energy through absorption. A calorimeter consists of a grid of detector modules (cells) that each individually measure deposited energy



Analysis – Example from High Energy Physics

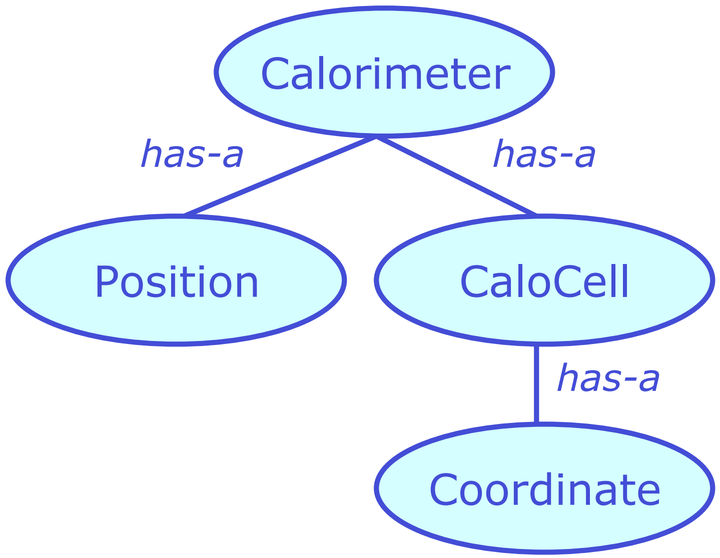
- First attempt to identify objects in data processing model and their containment hierarchy
  - Calorimeter global position and cell coordinates are not physical objects but separate logical entities so we make separate classes for those too





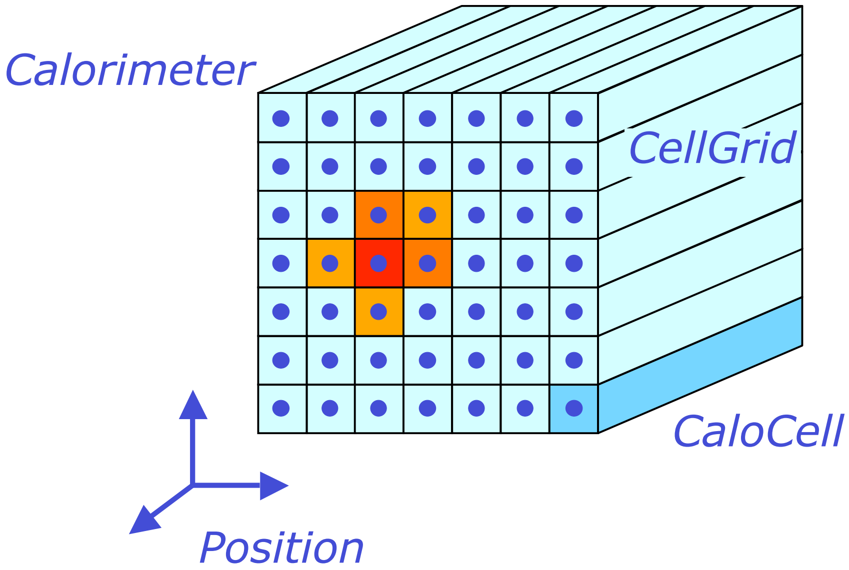
Analysis – Example from High Energy Physics

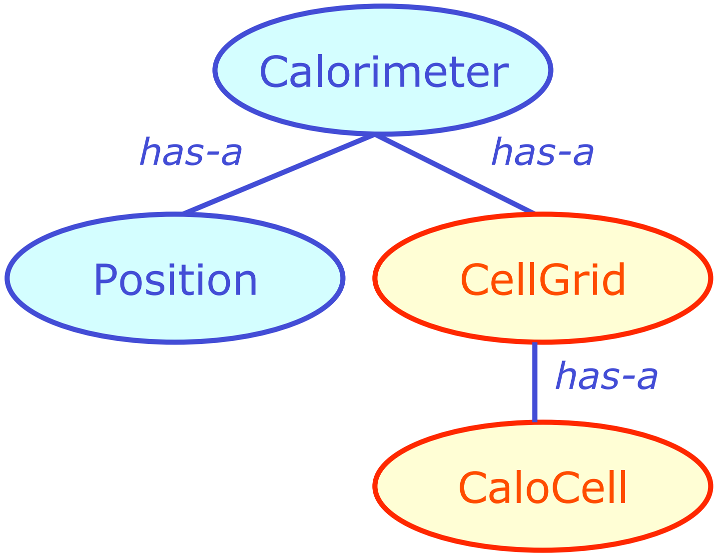
- Key Analysis sanity check – Can we describe what each object ***is***, in addition to what it does?
  - Answer yes



Analysis – Example from High Energy Physics

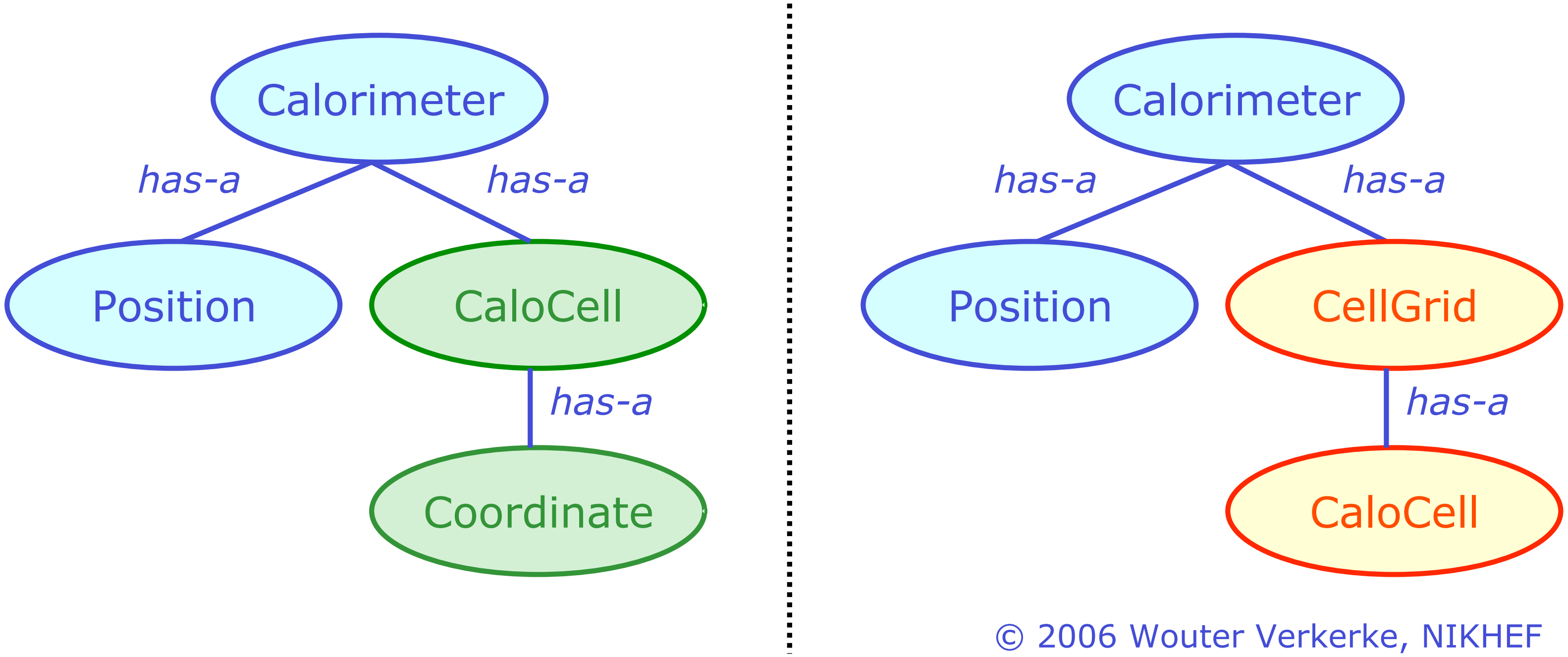
- Iterating the design – are there other/better solutions?
  - Remember 'strong cohesion' and 'loose coupling'
  - Try different class decomposition, moving functionality from one class to another
- Example of alternative solution
  - We can store the `CaloCells` in an intelligent container class `CellGrid`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





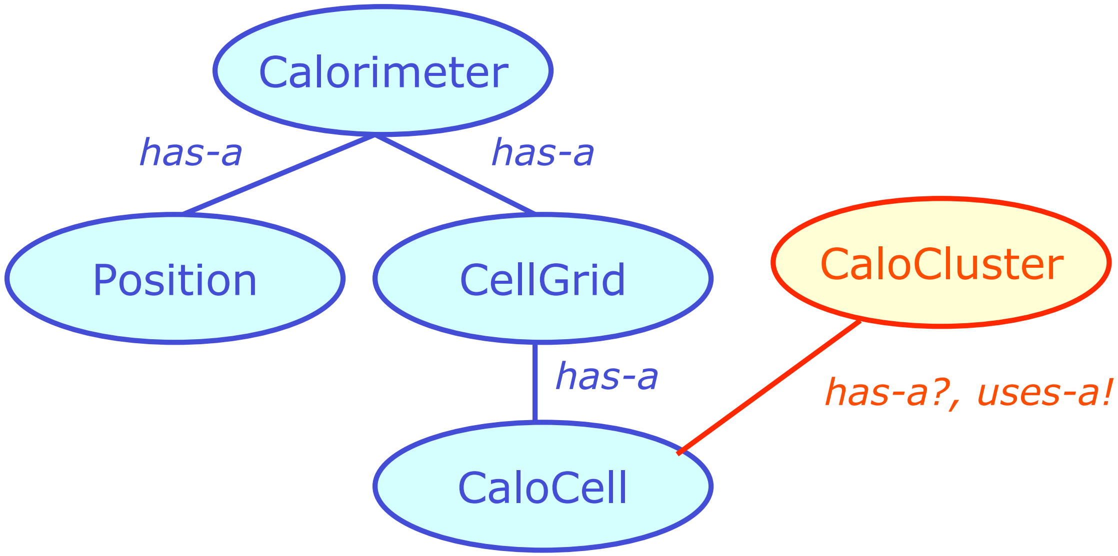
Analysis – Example from High Energy Physics

- 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



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

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)

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

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**

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**

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() ;
```

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 ; }

}
```

const version here is essential, otherwise code below will fail

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

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` $\Rightarrow$ `c.assign(a.multiply(b)) ;`

- Solution: C++ operator overloading

Operator overloading

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

What you write

What the compiler does

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

→

```
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…

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 ;
} ;
```

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

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 ;
   _array = new int[other._len] ;
   // Refill array here
   return *this ;
}
```

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

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

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)

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 ;
}
```

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

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() { 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

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() { 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**

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() { 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

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() { 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

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)

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

    delete[] _s ;                       // release orig memory

    _s = newstr ;                       // install new buffer
    return *this ;
  }
} ;
```

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
}
```

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

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` return 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) ;
}
```

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] ;
}
```

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 = phoneBook["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

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 for `classA` to `classB`

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

- Example implementation for class String

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

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) ;
}
```

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 we can define a dedicated operator that is ***computationally more efficient*** we should still implement it. The compiler will use the dedicated operator instead

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

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)`

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 the private parts of a function
  - 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&) ;
}
```

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] ;  // private members
  strcpy(s._buf,buf) ;          // of String s
  return is ;
}
```

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 ;
}
```

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()) ;
```

5 Standard Library – Using I/O streams

Introduction

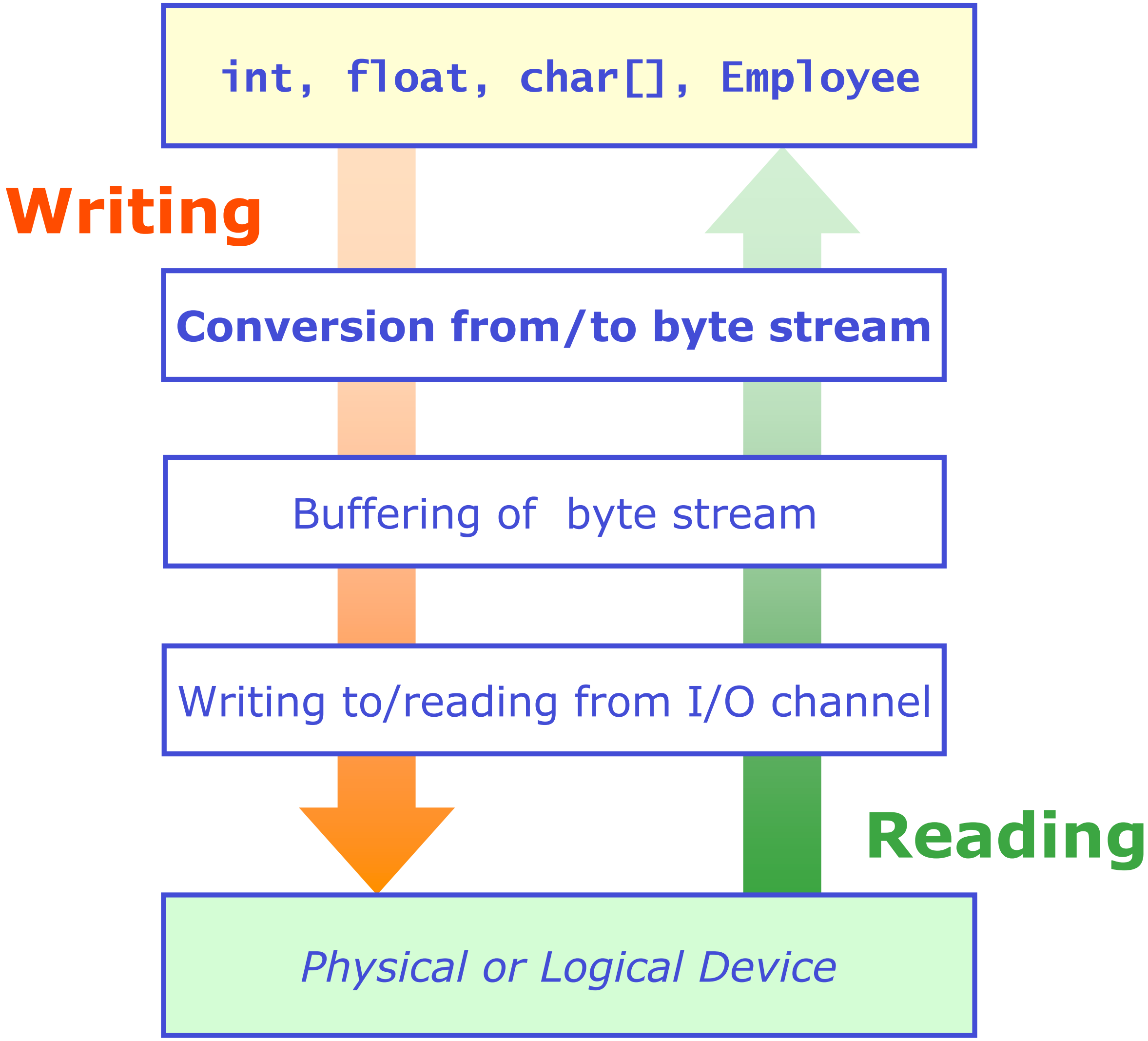
- The Standard Library organizes all kinds of I/O operations through a standard class '**`iostream`**'
- We've already used class `iostream` several types through the objects `cin` and `cout`

```
#include <iostream>
using namespace std ;

int main() {
  double x;

  // Read x from standard input
  cin >> x ;

  // Write x to standard output
  cout << "x = " << x << endl ;

  return 0 ;
}
```

- We will now take a better look at how streams work

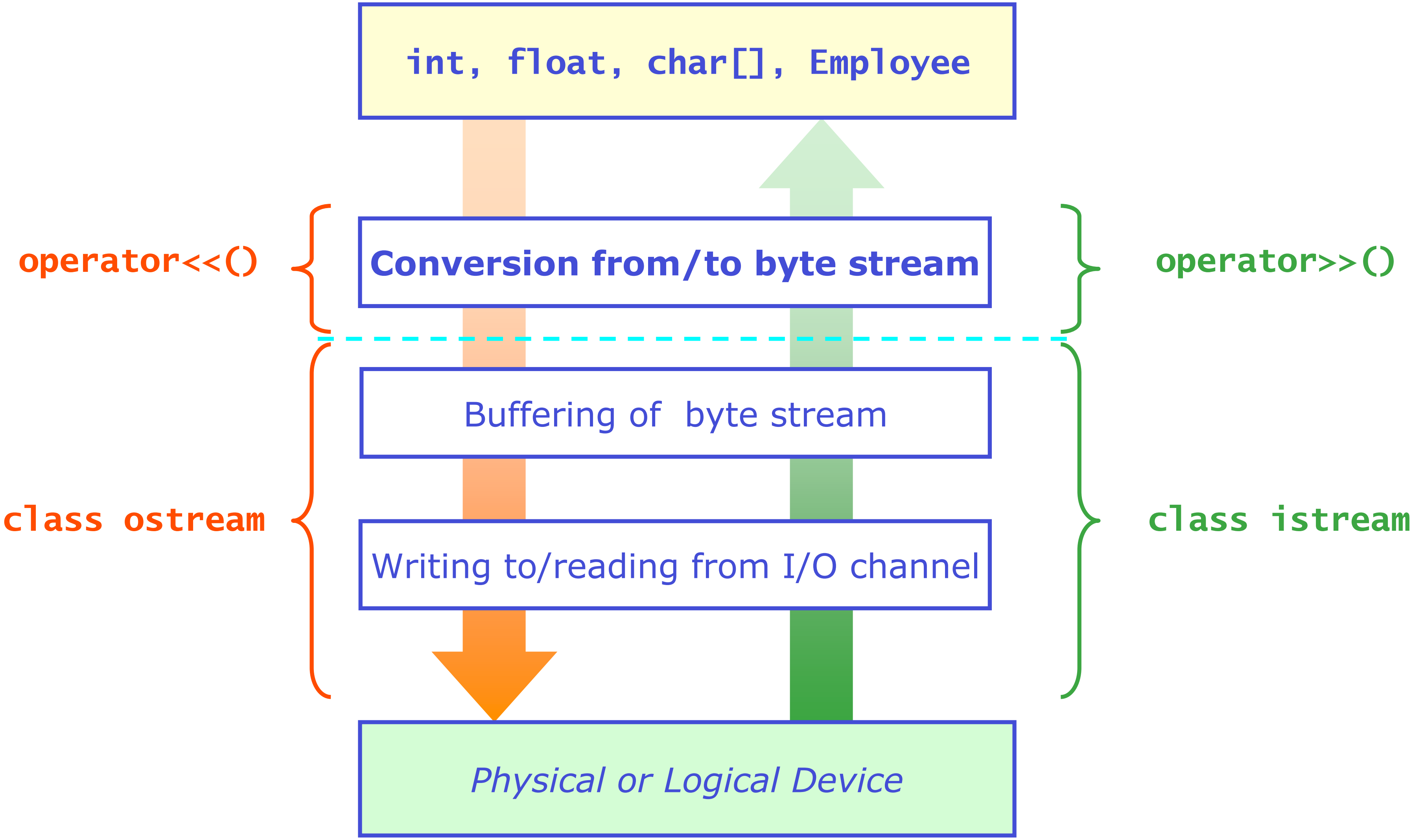
A look behind the scenes

- I/O in C++ involves three distinct steps



I/O classes and operators in C++

- Operators<<(), >>() do step 1, classes `istream`, `ostream` do step 2



Stream object in the Standard Library

- Stream classes in Standard Library

Include file	*Logical or physical device*	Input	Output	Both
<iostream>	Generic (e.g.terminal)	**istream**	**ostream**	**iostream**
<fstream>	File	**ifstream**	**ofstream**	**fstream**
<sstream>	`std::string`	**istringstream**	**ostringstream**	**stringstream**

Direction of byte stream: Input, Output, Both

- Standard Library stream classes also implement all operators to convert built-in types to byte streams
  - Implemented as member operators of stream class
  - Example: `ostream::operator<<(int) ;`

- Standard Library also provides three global stream objects for 'standard I/O'
  - `istream` object `cin` for 'standard input'
  - `ostream` objects `cout,cerr` for 'standard output','standard error'

Using streams without operators >>(),<<()

- Streams provide several basic functions to read and write bytes
  - Block operations

```
char buf[100] ;
int count(99) ;

// read 'count' bytes from input stream
cin.read(buf, count) ;

// write 'count' bytes to output stream
cout.write(buf,count) ;
```

Using streams without operators >>(),<<()

- Streams provide several basic functions to read and write bytes
  - Line oriented operations

```
// read line from stdin up to and including the newline char
cin.get(buf,100) ;

// read line from std up to newline char
cin.getline(buf,100) ;

// read line up to and including ‘:’
cin.get(buf,100,’:’) ;

// read single character
cin.get(c ) ;

// write buffer up to terminating null byte
cout.write(buf,strlen(buf)) ;

// write single character
cout.put(c ) ;
```

How is the stream doing?

- Member functions give insight into the *state* of the stream

Function	**Meaning**
`bool good()`	Next operation *might* succeed
`bool eof()`	End of input seen
`bool fail()`	Next operation will fail
`bool bad()`	Stream is corrupted

- Example – reading lines from a file till the end of the file

```
ifstream ifs(“file.txt”) ;
char buf[100] ;

// Loop as long as stream is OK
while(!ifs.fail()) {
   ifs.getline(buf,100) ;

   // Stop here if we have reached end of file
   if (ifs.eof()) break ;

   cout << “just read ‘” << buf << “’” << endl ;
}
```

Some handy abbreviations

- Streams overload operator `void*()` to return `!fail()`
  - Can shorten preceding example to

```
while(ifs) { // expanded to while(ifs.operator void*())
   ifs.getline(buf,100) ;
   if (ifs.eof()) break ;
   cout << "just read '" << buf << "'" << endl ;
}
```

- Also return value of `getline()` provides similar information
  - Returns true if stream is good() and stream is not at eof() after operation

```
while(ifs.getline(buf,100)) {
  cout << "just read '" << buf << "'" << endl ;
}
```

Using stream operators

- The next step is to use the streaming operators instead of the 'raw' IO routines
  - Encapsulation, abstraction → let objects deal with their own streaming
- Solution: use `operator>>()` instead of `getline()`

shoesize.txt

```
Bjarne 42
Leif 47
Thor 52
```

```
ifstream ifs("shoesize.txt") ;
string name ;
int size ;

while(ifs >> name >> size) {
  cout << name << " has  shoe size " << size << endl ;
}
```

Using stream operators

- Remember: syntax of stream operators is like that of any other operator

```
string name ;
int size
cin >> name >> size
;
```

```
string name ;
int size
cin.operator>>(   cin.operator>>(name),   size) ;
```

- For all built-in types
  - `operator<<(ostream,TYPE)` and `operator>>(istream,TYPE)` are implemented as member functions of the streams
  - Special case: `operator<<(const char*)` and `operator>>(char*)` read and write `char[]` strings

Parsing input – some fine points

- Delimiters
  - How does the text line

    `Bjarne Stroustrup 42`

    map on to the statement

    `cin >> firstName >> lastName >> shoeSize`

  - Because each `operator()>>` stops reading when it encounter 'white space'
  - White space is 'space','tab','vertical tab','form feed' and 'newline'
  - White space between tokens is automatically 'eaten' by the stream

- Reading string tokens
  - Be careful using `char[]` to read in strings: `operator>>(const char*)` does not know your buffer size and it can overrun!
  - Better to use `class string`

Formatting output of built-in types

- For built-in types streams have several functions that control formatting
  - Example: manipulating the base of integer output

```
cout.setf(ios_base::oct,ios_base::basefield) ; // set octal
cout <<  1234 << endl ; // shows '2322'

cout.setf(ios_base::hex,ios_base::basefield) ; // set hex
cout <<  1234 << endl ; // shows '4d2'
```

  - But it is often inconvenient to use this as calling formatting function interrupt chained output commands
- To accomplish formatting more conveniently streams have 'manipulators'
  - Manipulators are 'pseudo-objects' that change the state of the stream on the fly:

```
cout << oct << 1234 << endl << hex << 1234 << endl ;
// shows '2322' '4d2'
```

Overview of manipulators

- So manipulators are the easiest way to modify the formatting of built-in types
- What manipulators exist?
  - Integer formatting

Manipulator	Stream type	Description
`dec`	iostream	decimal base for integer
`hex`	iostream	hexadecimal base for integer
`oct`	iostream	octal base for integer
`[no]showpos`	iostream	show '+' for positive integers
`setbase(int n)`	iostream	base n for integer

  - Floating point formatting

Manipulator	Stream type	Description
`setprecision(int n)`	iostream	show n places after decimal point
`[no]showpoint`	iostream	[don't]show trailing decimal point
`scientific`	iostream	scientific format `x.xxexx`
`uppercase`	iostream	print `0XFF, nnExx`
`fixed`	iostream	format `xxxx.xx`

Manipulators – continued

- Alignment & general formatting

Manipulator	Stream type	Description
`left`	iostream	align left
`right`	iostream	align right
`internal`	iostream	use internal alignment for each type
`setw(int n)`	iostream	next field width is n positions
`setfill(char c)`	iostream	set field fill character to c

- Miscellaneous

Manipulator	Stream type	Description
`endl`	ostream	put '\n' and flush
`ends`	ostream	put '\0' and flush
`flush`	ostream	flush stream buffers
`ws`	istream	eat white space
`setfill(char c)`	iostream	set field fill character to c

- Include `<iomanip>` for most manipulator definitions

Formatting output with manipulators

- Very clever, but how do manipulators work?
  - A manipulator is a ‘pseudo-object’ that modifies the state of the stream
  - More precisely: a manipulator is a *static member function* of the stream that takes a stream as argument, for example

```
class ostream {
  static ostream& oct(ostream& os) {
    os.setf(ios::oct,ios::basefield) ;
  }
}
```

  - The manipulator applies its namesake modification to the stream argument
  - You put manipulators in your print statement because class `ostream` also defines

```
operator<<(ostream&(*f)(ostream&)) {
  return f(*this) ;
}
```

    This operator processes any function that takes a single `ostream&` as argument and returns an `ostream`. The operator calls the function with itself as argument, which then causes the wanted operation to be executed on itself

Random access streams

- Streams tied to files and to strings also allow random access
  - Can move 'current' position for reading and writing to arbitrary location in file or string

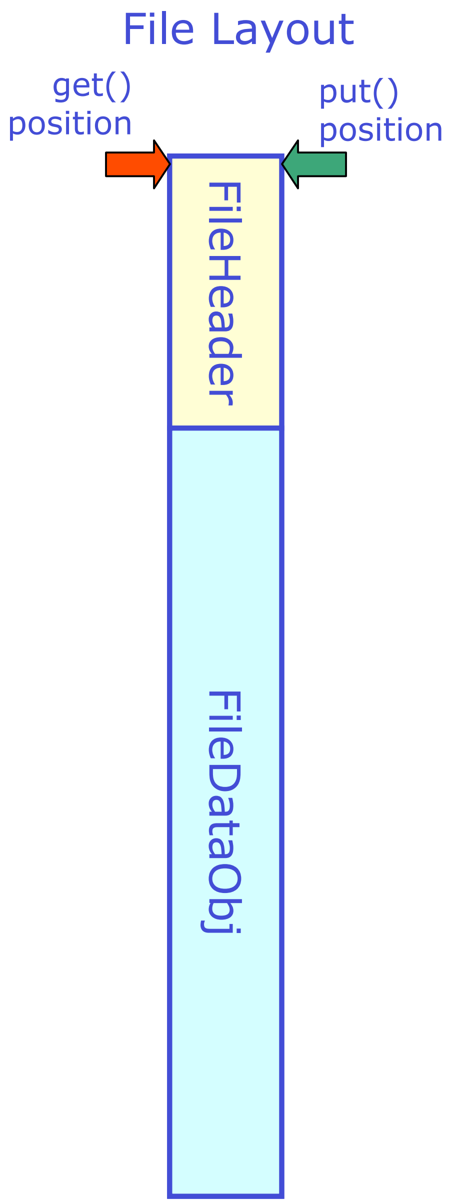
member function	stream	Description
`streampos tellg()`	input	return current location of 'get()' position
`seekg(streampos)`	input	set location of 'get()' position
`streampos tellp()`	output	return current location of 'put()' position
`seekp(streampos)`	output	set location of 'put()' position

  - Streams open for both input and output (`fstream,stringstream`) have all four methods, where `put()` and `get()` pointer can be in different positions

Random access streams

- Example use of `tell()`,`seek()`

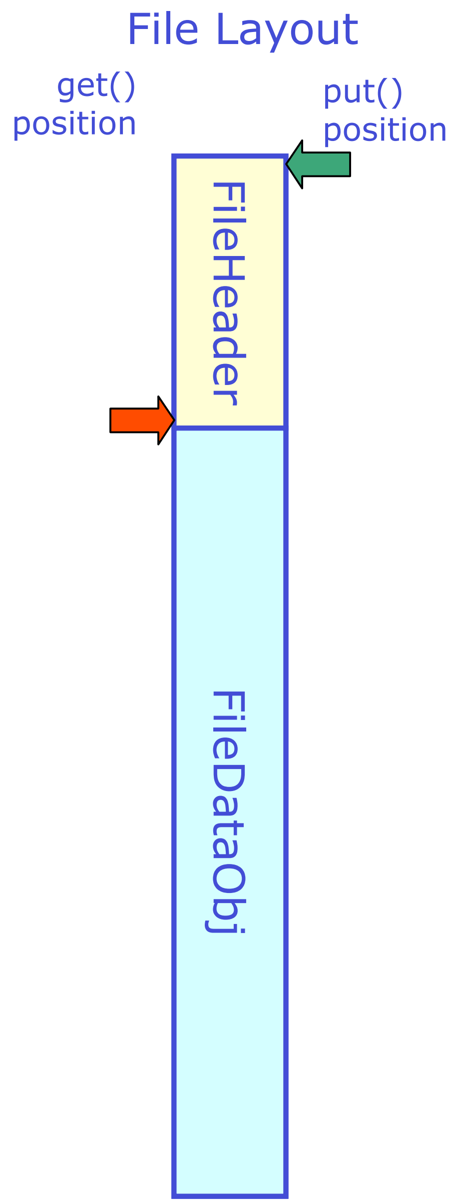
```
#include <fstream>

// Open file for reading and writing
fstream iofile("file.dat",ios::in|ios::out) ;
```



Random access streams

- Example use of `tell(),seek()`

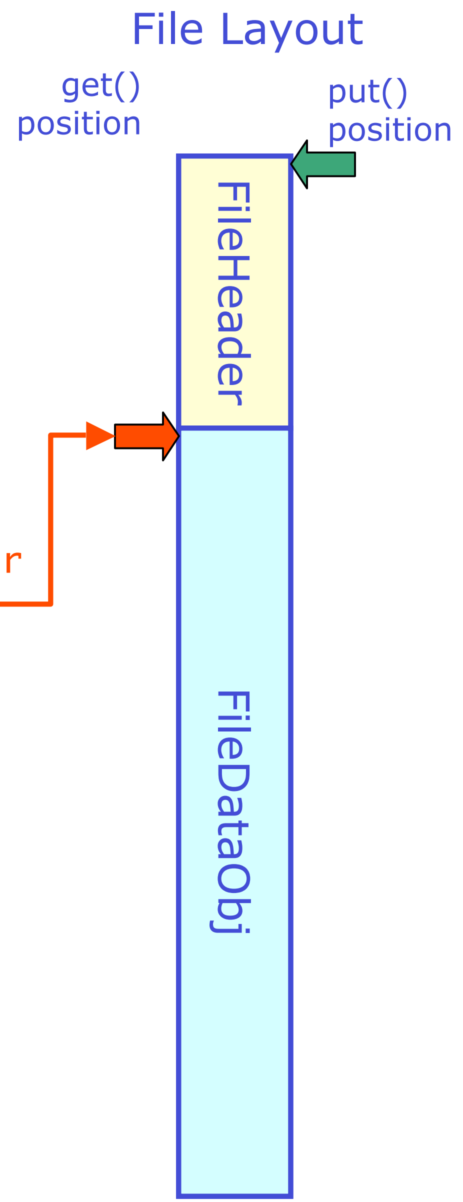
```
#include <fstream>

// Open file for reading and writing
fstream iofile("file.dat",ios::in|ios::out) ;

// Read in (fictitious) file header
FileHeader hdr ;
iofile  >> hdr ;
```



Random access streams

- Example use of `tell(),seek()`

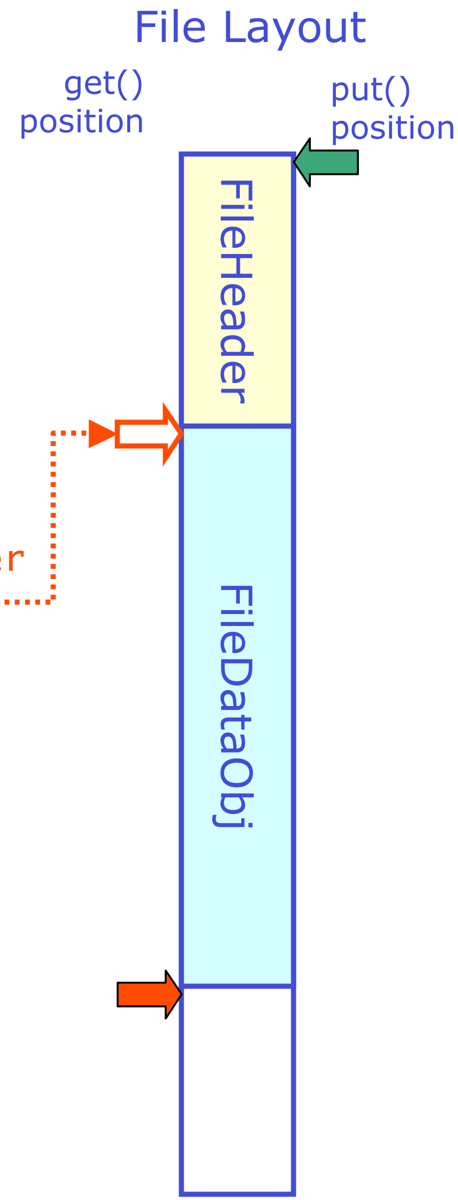
```
#include <fstream>

// Open file for reading and writing
fstream iofile("file.dat",ios::in|ios::out) ;

// Read in (fictitious) file header
FileHeader hdr ;
iofile  >> hdr ;

// Store current location of stream 'get()' pointer
streampos marker = ifs.tellg() ;
```



Random access streams

- Example use of `tell(),seek()`

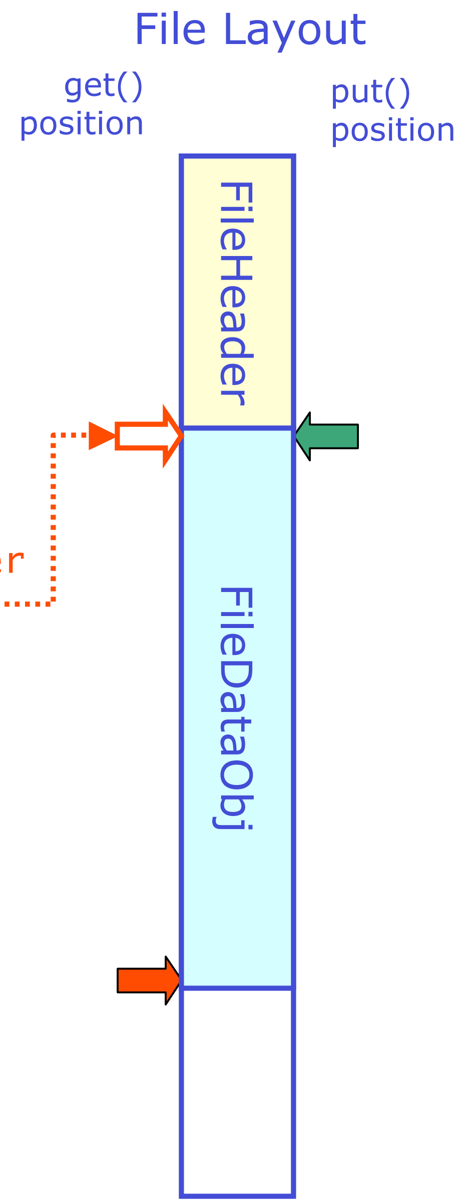
```
#include <fstream>

// Open file for reading and writing
fstream iofile("file.dat",ios::in|ios::out) ;

// Read in (fictitious) file header
FileHeader hdr ;
iofile  >> hdr ;

// Store current location of stream 'get()' pointer
streampos marker = ifs.tellg() ;

// Read (fictitious) file data object
FileDataObj fdo ;
iofile >> fdo ;
```



Random access streams

- Example use of `tell(),seek()`

```
#include <fstream>

// Open file for reading and writing
fstream iofile("file.dat",ios::in|ios::out) ;

// Read in (fictitious) file header
FileHeader hdr ;
iofile  >> hdr ;

// Store current location of stream 'get()' pointer
streampos marker = ifs.tellg() ;

// Read (fictitious) file data object
FileDataObj fdo ;
iofile >> fdo ;

// modify file data object

// Move current' location of stream
// 'put '()' pointer to marked position
ifs.tellp(marker) ;
```

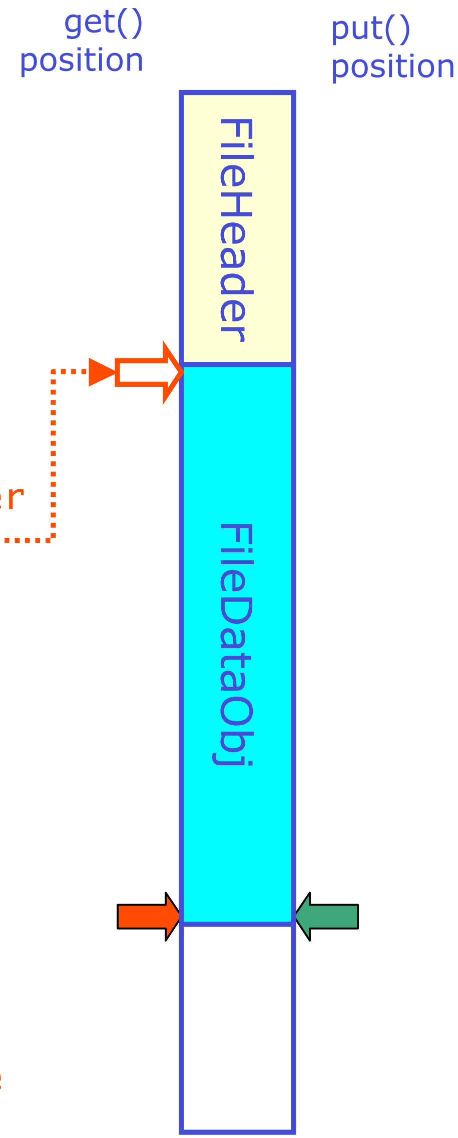


Random access streams

- Example use of `tell(),seek()`

```
#include <fstream>

// Open file for reading and writing
fstream iofile(“file.dat”,ios::in|ios::out) ;

// Read in (fictitious) file header
FileHeader hdr ;
iofile  >> hdr ;

// Store current location of stream ‘get()’ pointer
streampos marker = ifs.tellg() ;

// Read (fictitious) file data object
FileDataObj fdo ;
iofile >> fdo ;

// modify file data object

// Move current’ location of stream
// ‘put ‘()’ pointer to marked position
ifs.tellp(marker) ;

// Write modified object over old location in file
iofile << fdo ;
```

File Layout



Streaming custom classes

- You can stream custom classes by defining your matching `operator<<()`,`operator>>()` for those classes
  - Standard Library stream classes implement operators <<, >> as member functions for streaming of all basic types basic types
  - This is not an option for you as you can't modify the Standard Library classes
  - But in general, binary operators can be
    1. member of class `ostream(cout)`,
    2. member of your class, or
    3. be a global function.
  - Option 1) already ruled out
  - Option 2) doesn't work because class being read/written needs to be *right*most argument of operator, while as a member function it is by construction the *left* argument of the operator
  - Option 3) works: implement `operator<<` as global operator

Streaming custom classes

- For types that can be printed on a single line, overloading the operator<<, operator>> is sensible
  - Class string obvious example

```
String s(“Hello”) ;
cout << string << “ World” ;
```

```
String s(“Hello”) ;
cout.operator<<(operator<<(cout,string),”World”) ;
```

- For classes that read/write multi-line output, consider a separate function
  - operator>>,<< syntax for such cases potentially confusing: processing white space etc traditionally handled by stream not by operator
  - Example names: `readFromStream()`,`writeToStream()`

Implementing your own <<,>> operators

- Important: operators <<,>> need to return a reference to the input `ostream`, `istream` respectively
  - Essential for ability to chain << operations

```
cin >> a >> b ;
cout << a << b << c ;
```

- Example implementation for class string

```
ostream& operator<<(ostream& os, const String& s) {
   os << s._s ;
   return os ;
}

istream& operator>>(istream& is, String& s) {
   const int bufmax = 256 ;
   static char buf[256] ;
   is >> buf ;
   s = buf ;
   return is ;
}
```

Note: no const here
as String is modified

6 Generic Programming – Templates

Introduction to generic programming

- So far concentrated on definitions of objects as means of abstraction
- Next: Abstracting algorithms to be independent of the type of data they work with
- Naive – max()
  - Integer implementation

```
// Maximum of two values
int max(int a, int b) {
   return (a>b) ? a : b ;
}
```

  - (Naïve) real-life use

```
int m = 43, n = 56 ;
cout << max(m,n) << endl ; // displays 56 (CORRECT)

double x(4.3), y(5.6) ;
cout << max(x,y) << endl ; // displays 5 (INCORRECT)
```

Generic algorithms – the max() example

- First order solution – function overloading
  - Integer and float implementations

```
// Maximum of two values
int max(int a, int b) {
   return (a>b) ? a : b ;
}

// Maximum of two values
float max(float a, float b) {
   return (a>b) ? a : b ;
}
```

  - (Naïve) real-life use

```
int m = 43, n = 56 ;
cout << max(m,n) << endl ; // displays 56 (CORRECT)

double x(4.3), y(5.6) ;
cout << max(x,y) << endl ; // displays 5.6 (CORRECT)
```

Generic algorithms – the template solution

- Overloading solution works but not elegant
  - Duplicated code (always a sign of trouble)
  - We need to anticipate all use cases in advance

- C++ solution – a **template** function

```
template<class TYPE>
TYPE max(const TYPE& a, const TYPE& b) {
    return (a>b) ? a : b ;
}


int m = 43, n = 56 ;
cout << max(m,n) << endl ; // displays 56 (CORRECT)

double x(4.3), y(5.6) ;
cout << max(x,y) << endl ; // displays 5.6 (CORRECT)
```

Basics of templates

- A template function is function or algorithm for a generic TYPE
  - Whenever the compiler encounter use of a template function with a given TYPE that hasn't been used before the compiler will instantiate the function for that type

```
template<class TYPE>
TYPE max(const TYPE& a, const TYPE& b) {
    return (a>b) ? a : b ;
}

int m = 43, n = 56 ;
// compiler automatically instantiates max(int&, int&)
cout << max(m,n) << endl ; // displays 56 (CORRECT)

double x(4.3), y(5.6) ;
// compiler automaticallu instantiates max(float&, float&)
cout << max(x,y) << endl ; // displays 5.6 (CORRECT)
```

Basics to templates – assumptions on TYPE

- A template function encodes a generic algorithm but not a universal algorithm
  - TYPE still has to meet certain criteria to result in proper code
  - For example:

```
template<class TYPE>
TYPE max(const TYPE& a, const TYPE& b) {
    return (a>b) ? a : b ;
}
```

  assumes that `TYPE.operator>(TYPE&)` is defined

- Style tip: When you write a template spell out in the documentation what assumption you make (if any)

Basic templates – another example

- Here is another template function example

```
template <class TYPE>
void swap(TYPE& a, TYPE& b) {
    TYPE tmp = a ; // declare generic temporary
    a = b ;
    b = tmp ;
}
```

  - Allocation of generic storage space
  - Only assumption of this swap function: `TYPE::operator=()` defined
  - Since `operator=()` has a default implementation for all types this swap function truly universal
    - Unless of course a class declares `operator=()` to be private in which case no copies can be made at all

Template formalities

- Formal syntax of template function
  - Template declaration

```
template <class TYPE>
TYPE function_name(TYPE& t) ;
```

  - Template definition

```
template <class TYPE>
TYPE function_name(TYPE& t){
  // body
}
```

- What's OK
  - Multiple template classes allowed

```
template <class TYPE1, class TYPE2,…class TYPEN>
TYPE1 function_name(TYPE1&, TYPE2&,… TYPEN&) ;
```

  - Non-template function arguments allowed

```
template <class TYPE>
TYPE function_name(TYPE& t, int x, float y) ;
```

Template formalities

- And what's not
  - Template definitions must be in the global scope

```
int myFunction() {
   template <class T>        // ERROR - not allowed
     void myTemplFunc(T& t) ;
}
```

  - Template TYPE must appear in signature

```
template <class TYPE>
TYPE function_name(int i) // ERROR cannot overload
                          // on return type
```

  - Reason: function overloading by return type is not allowed

Templates, files and export

- Like for regular functions, template functions can be declared and defined in separates files
  - Template declaration file server to 'consumer code'
  - Template definition only offered once to compiler

- But one extra detail – template definitions in a separate file require the 'export' keyword

```
// swap.hh -- Declaration
template <class TYPE>
void swap(TYPE& a, TYPE& b) ;
```

```
// swap.cc -- Definition
export template <class TYPE>
void swap(TYPE& a, TYPE& b) {
  TYPE tmp = a ;
  a = b ;
  b = tmp ;
}
```

  - Reason: when templates definitions are in separate file logistics of instantiating all templates exactly once in a program consisting of many modules requires extra logistic support from compiler and linker. Export keyword tells compiler to make necessary preparations for these logistics

    NB: Keyword export is not supported by all compilers yet

Template specialization

- Sometimes you have a template function that is almost generic because
  - It doesn't work (right) with certain types.
    For example `max(const char* a, const char* b)`

```
template<class TYPE>
TYPE max(const TYPE& a, const TYPE& b) {
    return (a>b) ? a : b ; // comparing pointer not sensible
}
```

  - Or for certain types there is a more efficient implementation of the algorithm

- Solution: provide a *template specialization*
  - Can only be done in definition, not in declaration
  - Tells compiler that specialized version of function for given template should be used when appropriate

```
template<>
const char* max(const char* a, const char* b) {
    return strcmp(a,b)>0 ? a : b ; // Use string comparison instead
}
```

Template classes

- Concept of templates also extends to classes
  - Can define a template class just like a template function

```
template<class T>
class Triplet {
public:
   Triplet(T& t1, T& t2, T& t3) () ;
private:
   T _array[3] ;
}
```

- Class template mechanism allows to create generic classes
  - A generic class provide the same set of behaviors for all types
  - Eliminates code duplication
  - Simplifies library design
  - Use case per excellence: container classes (arrays, stacks etc…)

Generic container class example

- A generic stack example

```
template<class TYPE>
class Stack {
public:
  Stack(int size) : _len(size), _top(0) {     // constructor
      _v = new TYPE[_len] ;
   }
   Stack(const Stack<TYPE>& other) ;          // copy constructor
   ~Stack() { delete[] _v ; }

   void push(const TYPE& d) { _v[_top++] = d ; }
   TYPE pop() { return _v[--_top] ; }

   Stack<TYPE>& operator=(const Stack<TYPE>& s) ; // assignment

private:
   TYPE* _v ;
   int _len ;
   int _top ;

}
```

Assumptions on TYPE
-Default constructor
-Assignment defined

Using the generic container class

- Example using Stack

```
void example() {

   Stack<int> s(10) ; // stack of 10 integers
   Stack<String> t(20) ; // stack of 20 Strings

   s.push(1) ;
   s.push(2) ;
   cout << s.pop() << endl ;

   // OUTPUT '2'

   t.push("Hello") ; // Exploit automatic
   t.push("World") ; // const char* → String conversion

   cout << t.pop() << " " << t.pop() << endl ;

   // OUTPUT  'World Hello'

}
```

Non-class template parameters

- You can also add non-class parameters to a template declaration, e.g.
  - parameter must be constant expression (template is instantiated at compile time!)

```
template<class T, int DIM>
class Vector {
public:
   Vector() ;
   Vector(T[] input) ;
private:
   T _array[DIM] ;
}
```

  - Advantage: avoid dynamic memory allocation (runtime performance penalty)

```
void example() {
  Vector<double,3> ThreeVec ;
  Vector<double,4> FourVec ;
}
```

Template default parameters

- Default values for template parameters are also allowed

```
template<class T=double, int DIM=3>
class Vector {
public:
   Vector(T[] input) ;
private:
   T _array[DIM] ;
}
```

```
void example() {
  Vector          ThreeVecDouble ;
  Vector<int>     ThreeVecInt ;
  Vector<float,4> FourVecFloat ;
}
```

Containment, composition

- No real limit on complexity of template constructions
  - Containment

```
template<class TYPE>
class A {
private:
  B<TYPE> member1 ;        // OK – generic template member
  C<int> member2 ;         // OK – specific template member
  D member3 ;             // OK – non-template member
public:
  A(args) : B(args),C(args),D(args) {} // initialization
} ;
```

  - Composition

```
template<class TYPE> class Vec { … } ; // Vector container
template<class TYPE> class Arr { … } ; // Array container
template<class TYPE> class Sta { … } ; // Stack container

Vec<String> c1 ;             // Vector of Strings
Vec<Arr<String> > c2 ;       // Vector of Arrays of Strings
Vec<Arr<Sta<String> > > c3 ; // Vector of Arrays of Stacks of Strings
Vec<Vec<Vec<String> > > c4 ; // Vector of Vectors of Vectors of Strings
```

Note extra space to distguish from operator>>

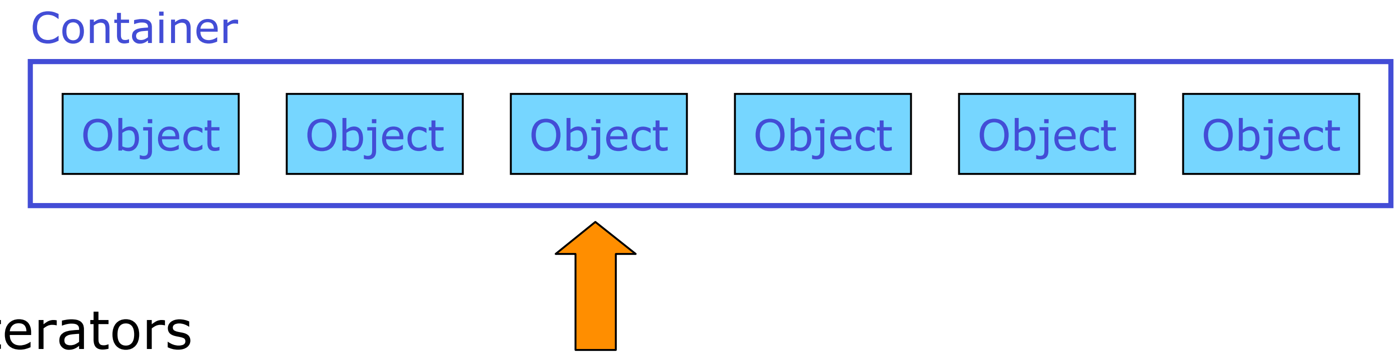
7 Standard Library II the Template Library

Introduction to STL

- STL = The **S**tandard **T**emplate **L**ibrary
  - A collection of template classes and functions for general use
  - Started out as experimental project by Hewlett-Packard
  - Now integral part of ANSI C++ definition of 'Standard Library'
  - Excellent design!
- Core functionality – Collection & Organization
  - Containers (such as lists)
  - Iterators (abstract methods to iterate of containers)
  - Algorithms (such as sorting container elements)
- Some other general-purpose classes
  - classes string, complex, bits

Overview of STL components

- Containers
  - Storage facility of objects



- Iterators
  - Abstract access mechanism to collection contents
  - “Pointer to container element” with functionality to move pointer

- Algorithms
  - Operations (modifications) of container organization of contents
  - Example: Sort contents, apply operation to each of elements

STL Advantages

- STL containers are generic
  - Templates let you use the same container class with any class or built-in type
- STL is efficient
  - The various containers provide different data structures.
  - No inheritance nor virtual functions are used (we'll cover this shortly).
  - You can choose the container that is most efficient for the type of operations you expect
- STL has a consistent interface
  - Many containers have the same interface, making the learning curve easier
- Algorithms are generic
  - Template functions allow the same algorithm to be applied to different containers.
- Iterators let you access elements consistently
  - Algorithms work with iterators
  - Iterators work like C++ pointers
- Many aspects can be customized easily

Overview of STL containers classes

- Sequential containers (with a defined order)
  - `vector`
  - `list`
  - `deque` (**d**ouble-**e**nded **que**ue)

  Fundamental container implementations
  with different performance tradeoffs

  - `stack`
  - `queue`
  - `priority_queue`

  Adapters of fundamental containers
  that provide a modified functionality

- Associative containers (no defined order, access by key)
  - `set`
  - `multiset`
  - `map`
  - `multimap`

Common container facilities

- Common operations on fundamental containers
  - **insert** – Insert element at defined location
  - **erase** – Remove element at defined location

  - **push_back** – Append element at end
  - **pop_back** – Remove & return element at end

  - **push_front** – Append element at front
  - **pop_front** – Remove & return element at front

  - **at** – Return element at defined location (with range checking)
  - **operator[]** – Return element at defined location (no range checking)

  - Not all operations exist at all containers (e.g. push_back is undefined on a set as there is no 'begin' or 'end' in an associative container)

Vector <vector>

- Vector is similar to an array

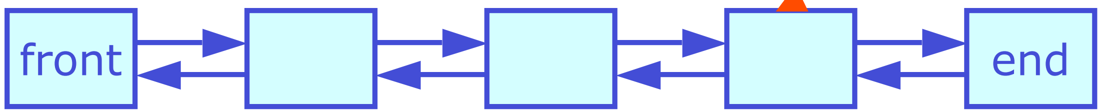
  - Manages its own memory allocation
  - Initial length at construction, but can be extended later
  - Elements initialized with default constructor
  - **Offers fast random access to elements**
  - Example

```
#include <vector>
vector<int> v(10) ;

v[0] = 80 ;
v.push_back(70) ; // creates v[10] and sets to 11

vector<double> v2(5,3.14) ; // initialize 5 elements to 3.14
```

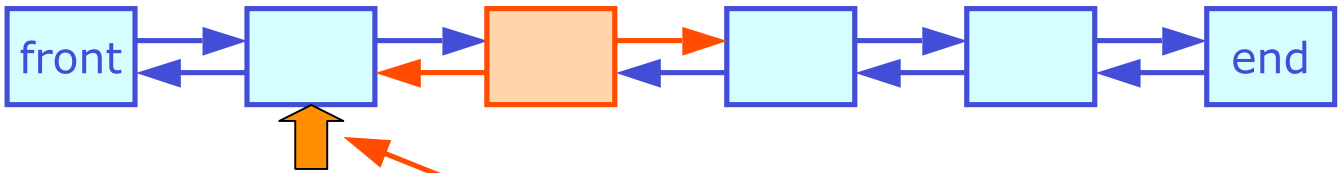
List <list>

```
Template<class T>
Struct ListElem {
  T elem ;
  ListElem* prev ;
  ListElem* next ;
}
```

- Implemented as doubly linked list



  - Fast insert/remove of in the middle of the collection



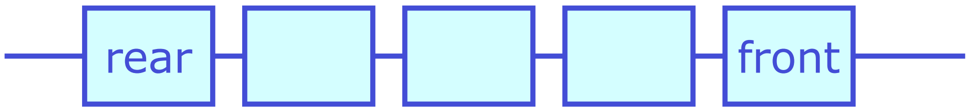
  - **No random access**
  - Example

iterator 'pointer' in collection

```
#include <list>
list<double> l ;
l.push_front(30.5) ; // append element in front
l.insert(somewhere,47.5) ; // insert in middle
```

Double ended queue <deque>

- Deque is sequence optimized for insertion at both ends
  - *Inserting at ends* is as efficient as `list`
  - *Random access* of elements efficient like `vector`

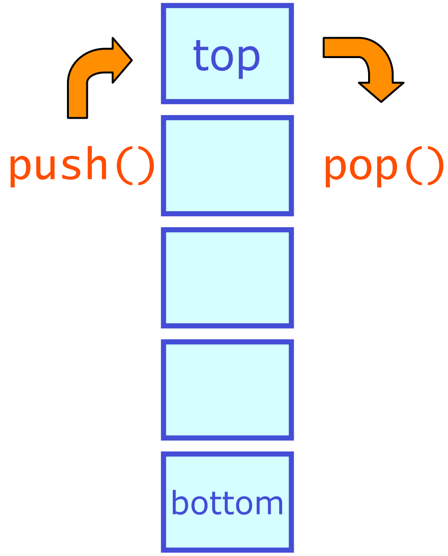


- Example of deque's in real life
  - String of beads
  - Deck of cards
  - Train

"Deque, it rhymes with 'check' (B. Stroustrup)"

Stack <stack>

- A stack is an adapter of deque
  - It provides a restricted view of a deque
  - Can only insert/remove elements elements at end (‘top’ in stack view’)
  - No random access

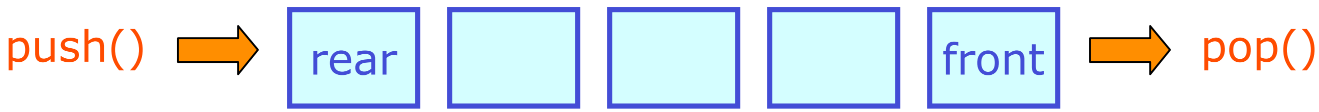


- Example

```
void sender() {
   stack<string> s ;
   s.push(“Aap”) ;
   s.push(“Noot”) ;
   s.push(“Mies”) ;
   receiver(s) ;
}
void receiver(stack<int>& s) {
   while(!s.empty()) cout << s.pop() << “ “ ;
}

// output “Mies Noot Aap”
```

Queue <queue>

- A queue is another adapter of deque
  - It provides a different restricted view of a deque
  - Can only insert elements at back, remove elements from front
  - No random access

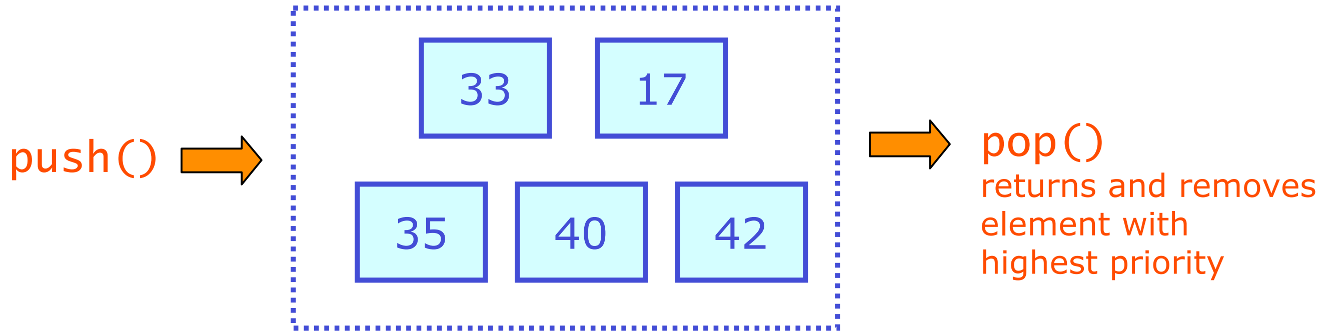


- Plenty of real-life applications
  - Ever need anything at city hall? Take a number!
  - Example implementation

```
void sender() {
   queue<string> s ;
   s.push(“Aap”) ; s.push(“Noot”) ;s.push(“Mies”) ;
   receiver(s) ;
}
void receiver(stack<int>& s) {
   while(!s.empty()) cout << s.pop() << “ “ ;
}
// output “Aap Noot Mies” (reverse compared to stack)
```

Priority_queue <queue>

- Like queue with priority control
  - In priority queue pop() returns element with highest rank, rank determined with operator<()
  - Also a container adapter (by default of vector)




```
void sender() {
   priority_queue<int> s ;
   s.push(10) ; s.push(30) ;s.push(20) ;
   receiver(s) ;
}
void receiver(stack<int>& s) {
   while(!s.empty()) cout << s.pop() << “ “ ;
}
// output “30 20 10”
```

Sequential container performance comparison

- Performance/capability comparison of sequential containers

		Insert/remove element		
	index []	In Middle	At Front	At Back
vector	const	O(n) +		const +
list		const	const	const
deque	const	O(n)	const	const
stack				const +
queue			const	const +
prio_que			O(logN)	O(logN)

  - const : *constant CPU cost*
  - O(n) : *CPU cost proportional to Nelem*
  - O(logN) : *CPU cost proportional to log(Nelem)*

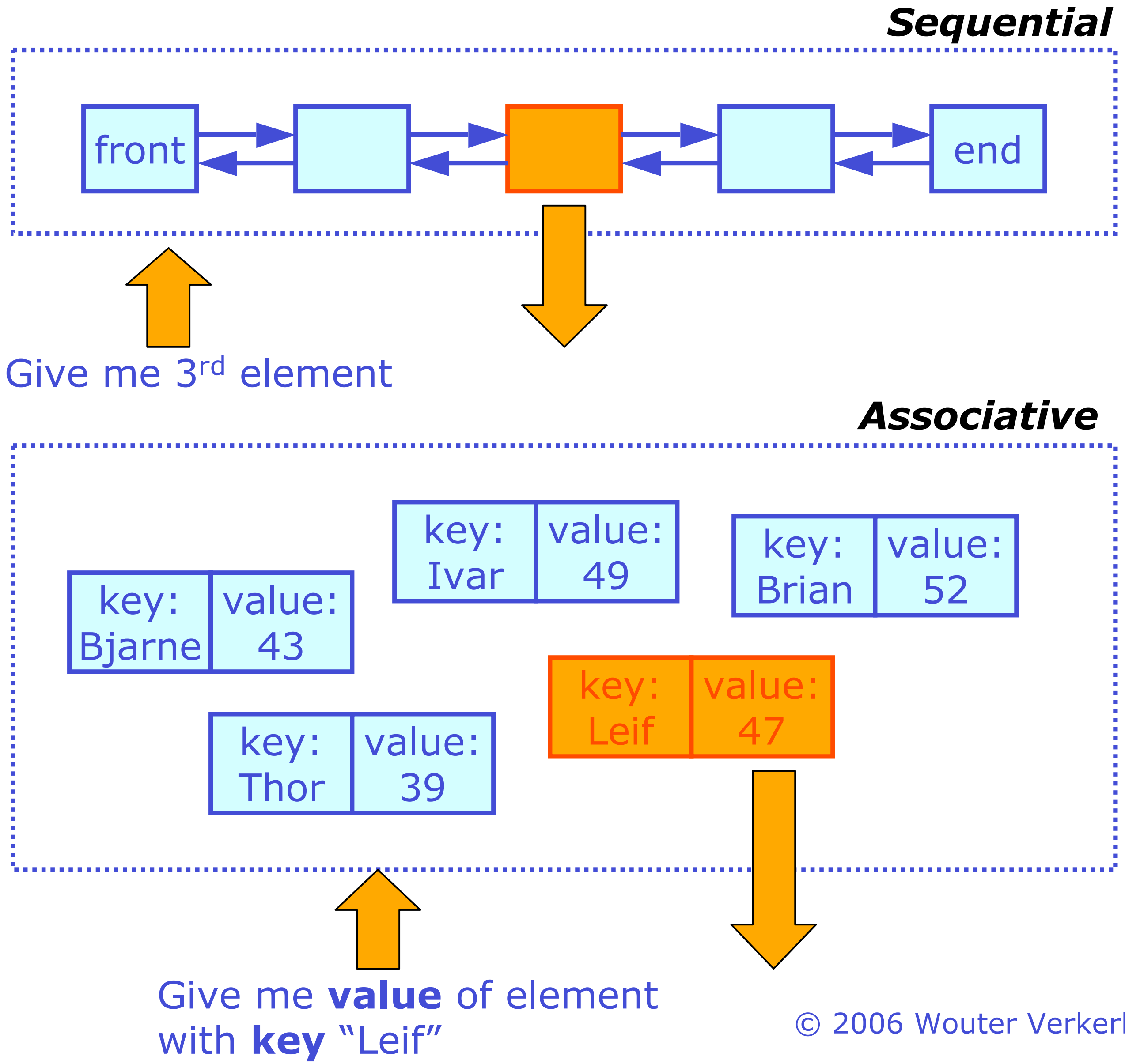
Sequential versus associative containers

- So far looked at several forms of *sequential* containers
  - Defining property: storage organization revolves around *ordering*: all elements are stored in a user defined order
  - Access to elements is always done by relative or absolute position in container
  - Example:

```
vector<int> v ;
v[3] = 3rd element of vector v

list<int> l ;
double tmp = *(l.begin()) ; // 1st element of list
```

- For many types of problems *access by key* is much more natural
  - Example: Phone book. You want to know the phone number (=value) for a name (e.g. 'B. Stroustrup' = key)
  - You don't care in which order collection is stored as you never retrieve the information by positional reference (i.e. you never ask: give me the 103102nd entry in the phone book)
  - Rather you want to access information with a 'key' associated with each value
- Solution: the **associative container**

Sequential versus associative containers



Pair <utility>

- Utility for associative containers – stores a key-value pair

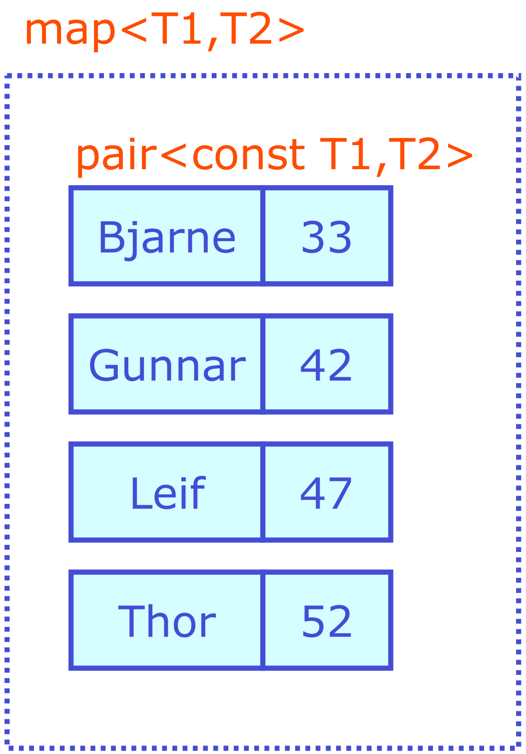
```
template<type T1, type T2>
struct pair {
   T1 first ;
   T2 second ;
   pair(const T1&, const T2&) ;
} ;

template<type T1, type T2>
pair<T1,T2> make_pair(T1,T2) ; // exists for convenience
```

  – Main use of pair is as input or return value

```
pair<int,float> calculation() {
   return make_pair(42,3.14159) ;
}
int main() {
   pair<int,float> result = calculation() ;
   cout << "result = " << pair.first
        << " " << pair.second << endl ;
}
```

Map <map>

- Map is an associative container
  - It stores pairs of *const* keys and values
  - Elements stored in ranking by keys (using `key::operator<()`)
  - **Provides direct access by key**
  - Multiple entries with same key prohibited



Map <map>

- Map example

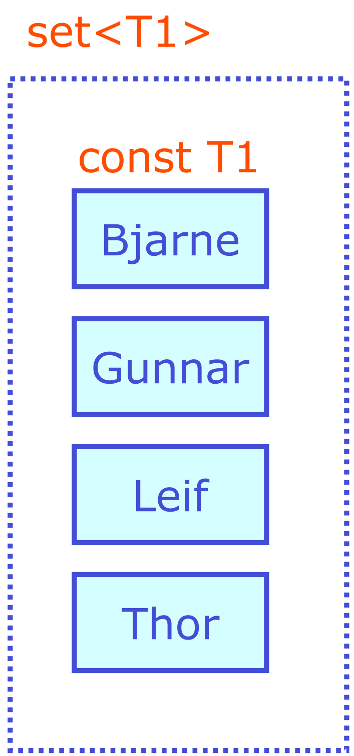
```
map<string,int> shoeSize ;

shoeSize.insert(pair<string,int>("Leif",47)) ;
showSize.insert(make_pair("Leif",47)) ;

shoeSize["Bjarne"] = 43 ;
shoeSize["Thor"] = 52 ;

int theSize = shoeSize["Bjarne"] ;     // theSize = 43
int another = shoeSize["Stroustrup"] ; // theSize = 0
```

  - If element is not found, new entry is added using default constructors

Set <set>

- A set is a map without values
  - I.e. it is just a collection of keys
  - No random access through []
  - Storage in ranked by key (<)
  - No duplicates allowed

```
set<string> people ;

people.insert("Leif") ;
people.insert("Bjarne") ;
people.insert("Stroustrup") ;
```



Multiset <set>, Multimap <map>

- Multiset
  - Identical to `set`
  - Except that multiple keys of the same value are allowed

- Multimap
  - Identical to `map`
  - Except that multiple keys of the same value are allowed

Taking a more abstract view of containers

- So far have dealt directly with container object to insert and retrieve elements
  - Drawback: Client code must know exactly what kind of container it is accessing
  - Better solution: provide an *abstract interface* to the container.
  - Advantage: the containers will provide the same interface (as far as possible within the constraints of its functionality)
  - Enhanced encapsulation – You can change the type of container class you use later without invasive changes to your client code

- STL abstraction mechanism for container access: **the iterator**
  - An iterator is *a pointer to an element in a container*
  - *So how is an iterator difference from a regular C++ pointer? – An iterator is aware of the collection it is bound to*.
  - *How to you get an iterator:* A member function of the collection will give it to you

Taking a more abstract view of containers

- Illustration of iterators vs C++ pointers

```
double v[10] ;

int i = 0 ;
double* ptr ;

ptr = &array[0] ;

while(i<10) {

  cout << *ptr << endl ;

  ++ptr ;
  ++i ;
}
```

Allocate C++ array of 10 elements

```
vector<double> v[10] ;


vector<double>::iterator iter ;

iter = v.begin() ;

while(iter!=v.end()) {

  cout << *iter << endl ;

  ++iter ;
}
```

Allocate STL vector of 10 elements

Taking a more abstract view of containers

- Illustration of iterators vs C++ pointers

```
double v[10] ;

int i = 0 ;
double* ptr ;

ptr = &array[0] ;

while(i<10) {

  cout << *ptr << endl ;

  ++ptr ;
  ++i ;
}
```

Allocate a pointer.
Also allocate an integer to keep track of when you're at the end of the array

```
vector<double> v[10] ;

vector<double>::iterator iter ;

iter = v.begin() ;

while(iter!=v.end()) {

  cout << *iter << endl ;

  ++iter ;
}
```

Allocate an STL iterator to a vector

Taking a more abstract view of containers

- Illustration of iterators vs C++ pointers

```
double v[10] ;

int i = 0 ;
double* ptr ;

ptr = &array[0] ;

while(i<10) {

  cout << *ptr << endl ;

  ++ptr ;
  ++i ;
}
```

Make the pointer point to the first element of the array

```
vector<double> v[10] ;


vector<double>::iterator iter ;

iter = v.begin() ;

while(iter!=v.end()) {

  cout << *iter << endl ;

  ++iter ;
}
```

Make the iterator point to the first element of the vector

Taking a more abstract view of containers

- Illustration of iterators vs C++ pointers

```
double v[10] ;

int i = 0 ;
double* ptr ;

ptr = &array[0] ;

while(i<10) {

  cout << *ptr << endl ;

  ++ptr ;
  ++i ;
}
```

Check if you're at the end of your array

```
vector<double> v[10] ;


vector<double>::iterator iter ;

iter = v.begin() ;

while(iter!=v.end()) {

  cout << *iter << endl ;

  ++iter ;
}
```

Check if you're at the end of your vector

Taking a more abstract view of containers

- Illustration of iterators vs C++ pointers

```
double v[10] ;

int i = 0 ;
double* ptr ;

ptr = &array[0] ;

while(i<10) {

  cout << *ptr << endl ;

  ++ptr ;
  ++i ;
}
```

Access the element the pointer is currently pointing to

```
vector<double> v[10] ;


vector<double>::iterator iter ;

iter = v.begin() ;

while(iter!=v.end()) {

  cout << *iter << endl ;

  ++iter ;
}
```

Access the element the iterator is currently pointing to

Taking a more abstract view of containers

- Illustration of iterators vs C++ pointers

```
double v[10] ;

int i = 0 ;
double* ptr ;

ptr = &array[0] ;

while(i<10) {

  cout << *ptr << endl ;

  ++ptr ;
  ++i ;
}
```

Modifiy the pointer to point to the next element in the array

```
vector<double> v[10] ;


vector<double>::iterator iter ;

iter = v.begin() ;

while(iter!=v.end()) {

  cout << *iter << endl ;

  ++iter ;

}
```

Modify the iterator to point to the next element in the array

Containers Iterators – A closer look at the formalism

```
// Iterator loop
vector<int> v(10) ;
vector<int>::iterator iter ;
for (iter=v.begin() ;
     iter!=v.end() ;
          ++iter) {
  *iter = 0 ;
}
```

Declaration
container<T>::iterator

begin() returns iterator
pointing to start of container

operator++
moves iterator one
element ahead

end() returns iterator pointing
to start of container

assignment between
iterators transfer
'current position'

dereference
operator*() returns
T& to pointed-to
element

Why iterators are a better interface

- Iterators hide the structure of the container
  - Iterating over a vector, list or map works the same. You client code needs to make no (unnecessary assumptions) on the type of collection you're dealing with

```
// Iterator loop over a vector
vector<int> v(10) ;
int sum = calcSum(v.begin()) ;

int calcSum(vector<int>::iterator iter) {
  int sum(0) ;
  while(iter) {
    sum += *iter ;
    ++iter ;
  }
  return sum ;
}
```

```
// Iterator loop over a list
list<int> l ;
int sum = calcSum(l.begin()) ;

int calcSum(list<int>::iterator iter) {
  int sum(0) ;
  while(iter) {
    sum += *iter ;
    ++iter ;
  }
  return sum ;
}
```

Iterators and ranges

- A pair of iterators is also an efficient way to define subsets of containers
  - Example with multimap-based phone book

```
#include <map>

multimap<string,int> pbook ;
pbook.insert(pair<string,int>("Bjarne",00205726666)) ; // office phone
pbook.insert(pair<string,int>("Bjarne",00774557612)) ; // home phone
pbook.insert(pair<string,int>("Bjarne",0655765432)) ; // cell phone
pbook.insert(pair<string,int>("Fred",0215727576))  ; // office phone

multimap<string,int>::iterator iter
    begin=pbook.lower_bound("Bjarne"),
    end=pbook.upper_bound("Bjarne") ;

for(iter=begin ; iter!=end ; ++iter) {
    cout << iter->first << " " << iter->second << endl ;
}
```

  - Running iterator in standard way between bounds supplied by `lower_bound()` and `upper_bound()` accesses all elements with key "Bjarne"
  - No special mechanism needed to indicate subsets

Iterators and container capabilities

- Not all containers have the same capabilities
  - For example list does not allow random access (i.e. you cannot say: give me the 3rd element. You can say: give the next element with respect to the current iterator position, or the preceding element)
- Solution: STL provides multiple types of iterators with different capabilities
  - All iterators look and feel the same so you don't notice they're different, except that if a container can't do a certain thing the corresponding iterator won't let you do it either
  - Within a given set of functionality (e.g. only sequential access but no random access) all STL containers that provide that interface look and feel the same when access through iterators
- What classes of iterators exist:
  - Input, Ouput, Forward, Bidirectional, RandomAccess

Types of iterators

- Input iterator
  - Special iterator for input, for example from keyboard
  - Iterator allows to read input and must be incremented before next input is allowed

```
var = *iter++ ;
```

- Output iterator
  - Special iterator for output sequence, for example to standard output
  - Iterator allows to write and must be incremented before next output is allowed

```
*iter++ = var ;
```

- Forward iterator
  - Input and output are both allowed
  - Iteration must occur in positive increments of one

```
*iter = var ;
var = *iter ;
++iter ;
```

Types of iterators

- Bidirectional iterator
  - Can move forward and backward in steps of one

```
*iter = var ;
var = *iter ;
++iter ;
--iter ;
```

- Random access iterator
  - Can move with arbitrary step size, or move to absolute locations

```
*iter = var ;
var = *iter ;
++iter ;
--iter ;
iter[3] = var ;
iter += 5 ;
iter -= 3 ;
```

Containers and iterators

- Iterator functionality overview

Input Iterator →
Output Iterator → Forward Iterator → Bidirectional Iterator → Random Access Iterator

Less functionality → More functionality

- Iterators provided by STL containers

Container	Provided iterator
`vector`	random access
`deque`	random access
`list`	bidirectional
`(multi)set`	bidirectional
`(multi)map`	bidirectional
`stack`	none
`(priority_)queue`	none

Obtaining iterators

```
// Forward iterators
C.begin()
C.end()

// Reverse iterators
C.rbegin()
C.rend()
```

Iterators as arguments to generic algorithms

- Iterators allow generic algorithms to be applied to contains
  - Iterators hide structure of container → Access through iterator allows single algorithm implementation to operate on multiple container types
  - Examples

```
vector vec<int> vec(10) ;

// Shuffle vector elements in random order
random_shuffle(vec.begin(),vec.end()) ;

// Sort vector elements according to operator< ranking
sort(vec.begin(),vec.end()) ;

list l<string> l ;
// Sort list elements according to operator< ranking
sort(l.begin(),l.end()) ;
```

STL algorithms – sort

- Sorts elements of a sequential container
  - Uses `operator<()` for ranking
  - Needs RandomAccess iterators
- Example

```
vector<int> grades(100) ;
sort(grades.begin(),grades.end()) ; // sort all elements

vector<int>::iterator halfway = grades.begin()+50 ;
sort(grades.begin(),halfway) ; // sort elements [0,49]
```

- Notes
  - Pair of iterators is used to indicate range to be sorted
  - Range does not include element pointed to by upper bound iterator
    - Function `end()` returns a 'past the end' iterator so that the last element of the container is included in the range when `end()` is used as endpoint

STL algorithms – find

- Finds element of a certain value in a sequential container
  - Uses `operator==()` to establish matches
  - Expects ForwardIterator
  - Return value: iterator pointing to element, `end()` otherwise
- Example

```
list<int> l(10) ;
int value = 3 ;

// Find first occurrence
list<int>::iterator iter ;
iter = find(l.begin(),l.end(),value) ;
if (iter != l.end()) {
   // element found
}

// Find  remaining occurrences
iter = find(iter,l.end(),value)
while (iter != l.end()) {
   // element found
   iter = find(++iter,l.end(),value))
}
```

STL algorithm – for_each

- Calls function for each element in sequential container
  - Pass *call back function* to algorithm
  - Call back function should take single argument of type matching container, returning void
  - Expects ForwardIterator

- Example

```
// the call back function
void printRoot(float number) {
  cout << sqrt(number) << endl ;
}

// Execute the algorithm
vector<float> v(10) ;
for_each(v.begin(),v.end(),printRoot) ;

// Calls printRoot for each element of v,
// passing its value
```

STL algorithm – copy

- Copies (part of) sequential container to another sequential container
  - Takes two input iterators and one output iterator
- Example

```
list<int> l(10) ;
vector<int> v(10) ;

// copy from list to vector
copy(l.begin(), l.end(), v.begin()) ;

// copy from list to standard output
copy(l.begin(), l.end(), ostream_iterator<int>(cout,"\n")) ;
```

- Note on `ostream_iterator`
  - Construct output iterator tied to given ostream.
  - Optional second argument is printed after each object is printed

STL algorithm overview

- There are many algorithms!

accumulate	adjacent_difference	binary_search	copy	copy_backward
count	count_if	equal	equal_range	fill
fill_n	find	find_if	for_each	generate
generate_n	includes	inner_product	inplace_merge	iota
iter_swap	lexographical_compare	compare	lower_bound	make_heap
max	max_element	merge	min_element	mismatch
next_permutation	partial_sort_copy	partial_sum	partition	pop_heap
prev_permutation	push_heap	random_shuffle	remove	remove_copy
remove_copy_if	remove_if	replace	replace_copy	replace_copy_if
replace_if	reverse	reverse_copy	rotate	rotate_copy
search	set_difference	set_intersection	set_symmetric_difference	set_union
sort	sort_heap	stable_partition	stable_sort	swap
swap_ranges	transform	unique	nthe_element	partial_sort
unique_copy	upper_bound			

Modifying default behavior – Function objects

- STL container and algorithm behavior can easily be adapted using optional call back functions or function objects
  - Example from `for_each`

```
// the call back function
void printRoot(float number) {
  cout << sqrt(number) << endl ;
}

// Execute the algorithm
vector<float> v(10) ;
for_each(v.begin(),v.end(),printRoot) ;
```

- Mechanism generalized in several ways in STL
  - Customization argument not necessarily a function, but can also be a 'function object', i.e. anything that can be evaluated with `operator()`.
  - STL provides a set of standard function objects to apply common behavior modifications to algorithms and containers

Function objects

- Illustration of function object concept
  - A class with `operator()(int)` has the same call signature as `function(int)`

```
// the call back function

void printRoot(float number) {
  cout << sqrt(number) << endl ;
}
```

```
// Execute the algorithm
vector<float> v(10) ;
for_each(v.begin(),v.end(),
         printRoot) ;
```

```
// the function object
class printRoot {
public:
  operator()(float number) {
     cout << sqrt(number) << endl ;
  }
}
```

```
// Execute the algorithm
vector<float> v(10) ;
for_each(v.begin(),v.end(),
         printRoot()) ;
```

Template function objects

- Function objects can also be templated
  - Introduces ***generic*** function objects
  - `printRoot` is example of *unary* function object (takes 1 argument)
  - binary function objects are also common (2 arguments, returns `bool`)
    - To implement comparisons, ranking etc

```
// the template function object
template <class T>
class printRoot {
public:
  void operator()(T number) {
     cout << sqrt(T) << endl ;
  }
}
```

```
// Execute the algorithm on vector of float
vector<float> v(10) ;
for_each(v.begin(),v.end(),printRoot<float>()) ;
```

```
// Execute the algorithm on vector of int
vector<int> v(10) ;
for_each(v.begin(),v.end(),printRoot<int>()) ;
```

STL Function objects <functional>

- STL already defines several template function objects
  - Arithmetic: `plus, minus, divides,…`
  - Comparison: `less, greater, equal_to`
  - Logical: `logical_and, logical_or,…`

- Easy to use standard function objects to tailor STL algorithms
  - Example

```
vector<int> v(10) ;

// Default sort using int::operator<()
sort(v.begin(),v.end()) ;

// Customized sort in reversed order
sort(v.begin(),v.end(),greater<int>()) ;
```

  - Also used in some container constructors

```
bool NoCase(const string& s1, const string& s2) ;
map<string,int,NoCase> phoneBook ;
```

Numeric support in STL

- STL has some support for numeric algorithms as well
  - Won't mention most of them here except for one:

- Class complex
  - STL implements complex numbers as template

```
complex<double,double> a(5,3), b(1,7) ;
complex<double,double> c = a * b ;
```

  - Performance optimized template specializations exist for `<double,double>`, `<float,float>`, `<long double, long double>`
  - Default template argument is `float`

```
complex a(5,3), b(1,7) ;
complex c = a * b ;
```

  - Nearly all operators are implemented
    - complex * double, int * complex etc etc…

8 Inheritance & Polymorphism

Inheritance – Introduction

- Inheritance is
  - **a technique to build a new class based on an old class**
- Example
  - Class employee holds employee personnel record

```
class Employee {
public:
  Employee(const char* name, double salary) ;
  const char* name() const ;
  double salary() const ;
private:
  string _name ;
  double _salary ;
} ;
```

  - Company also employs managers, which in addition to being employees themselves supervise other personnel
    - Manager class needs to contain additional information: list of subordinates
  - Solution: make Manager class that ***inherits*** from Employee

Inheritance – Syntax

- Example of Manager class constructed through inheritance

Declaration of public inheritance

```
class Manager : public Employee {
public:
  Manager(const char* name, double salary,
          vector<Employee*> subordinates) ;
  list<Employee*> subs() const ;
private:
  list<Employee*> _subs ;
} ;
```

Additional data members in Manager class

Inheritance and OOAD

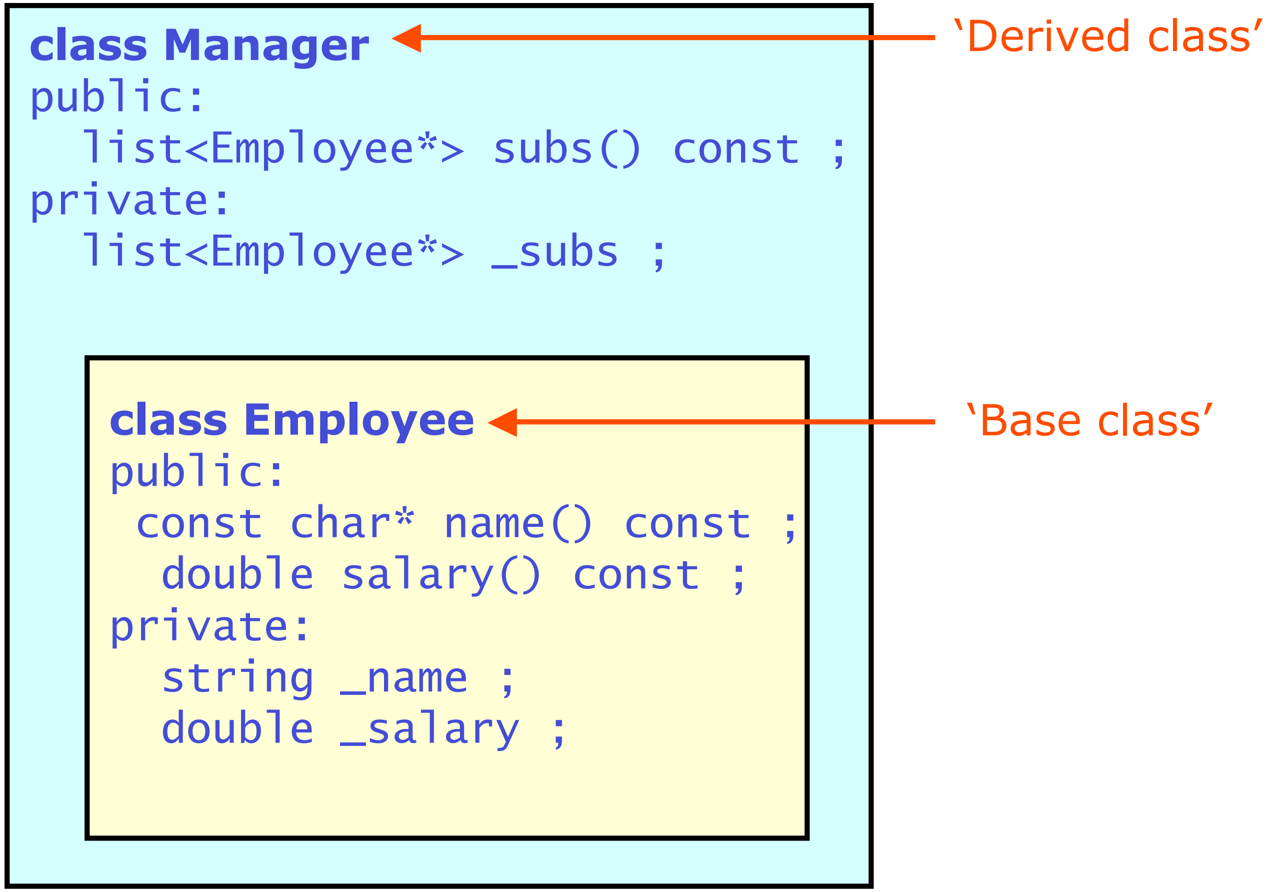
- Inheritance means: Manager **Is-A**n Employee
  - Object of class Manager can be used in exactly the same way as you would use an object of class Employee because
  - class Manager also has all data members and member functions of class Employee
  - Detail: examples shows '*public inheritance*' – Derived class inherits *public interface* of Base class

- Inheritance offers new possibilities in OO Analysis and Design
  - But added complexity is major source for conceptual problems
  - We'll look at that in a second, let's first have a better look at examples

Inheritance – Example in pictures

- Schematic view of Manager class

Terminology



Inheritance – Example in pictures

- Inheritance can be used recursively

Terminology

```
class Director
public:
   int numShares() const ;
private:
   int _numShares() ;

   class Manager
   public:
     list<Employee*> subs() const ;
   private:
     list<Employee*> _subs ;

      class Employee
      public:
       const char* name() const ;
        double salary() const ;
      private:
        string _name ;
        double _salary ;
```

'Derived class'

'Derived class' & 'Base class'

'Base class'

Inheritance – Using it

- Demonstration of Manager-IS-Employee concept

```
// Create employee, manager record
Employee* emp = new Employee("Wouter",10000) ;

list<Employee*>subs(1) ;
subs.push_back(wouter) ;

Manager* mgr = new Manager("Stan",20000,subs) ;


// Print names and salaries using
// Employee::salary() and Employee::name()
cout << emp->name() << endl ;    // prints Wouter
cout << emp->salary() << endl ;  // prints 10000

cout << mgr->name() << endl ;    // prints Stan
cout << mgr->salary() << endl ;  // prints 20000
```

Inheritance – Using it

- Demonstration of Manager-IS-Employee concept
  - A Pointer to a derived class is a pointer to the base class

```
// Pointer-to-derived IS Pointer-to-base
void processEmployee(Employee& emp) {
   cout << emp.name() << " : " << emp.salary() << endl ;
}

processEmployee(*emp) ;
processEmployee(*mgr) ; // OK Manager IS Employee
```

  - But there reverse is not true!

```
// Manager details are not visible through Employee* ptr
Employee* emp2 = mgr ; // OK Manager IS Employee
emp2->subs() ; // ERROR – Employee is not manager
```

OO Analysis and Design – 'Is-A' versus 'Has-A'

- How is an 'Is-A' relationship different from a 'Has-A' relationship
  - An Is-A relationship expresses inheritance (A is B)
  - An Has-A relationship expresses composition (A is a component of B)

a `Calorimeter` **HAS-A** `Position`

```
class Calorimeter {
public:
  Position& p() { return p ; }
private:
  Position p ;
} ;
```

```
Calorimeter calo ;
// access position part
calo.p() ;
```

An `Manager` **IS-An** `Employee`

```
class Manager :
       public Employee {
public:

private:
} ;
```

```
Manager mgr ;
// Use employee aspect of mgr
mgr.salary() ;
```

Inheritance – constructors, initialization order

- Construction of derived class involves construction of base object ***and*** derived object
  - Derived class constructor must call base class constructor
  - The base class constructor is executed *before* the derived class ctor
  - Applies to all constructors, *including the copy constructor*

```
Manager::Manager(const char* name, double salary,
                            list<Employee*>& l) :
   Employee(name,salary),
   _subs(l) {
   cout << name() << endl ; // OK - Employee part of object
}                           // is fully constructed at this
                            // point so call to base class
                            // function is well defined

Manager::Manager(const Manager& other) :
   Employee(other), // OK Manager IS Employee
   _subs(other._subs) {
   // body of Manager copy constructor
}
```

Inheritance – Assignment

- If you define your own assignment operator for an inherited class (e.g. because you allocate memory) you need to handle the base class assignment as well
  - Virtual function call mechanism invokes call to derived class assignment operator only.
  - You should call the base class assignment operator in the derived class assignment operator

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

  // Handle self assignment
   if (&other != this) return *this ;

   // Handle base class assignment
   Employee::operator=(other) ;

  // Derived class assignment happens here

  return *this ;
}
```

Inheritance – Destructors, call sequence

- For destructors the reverse sequences is followed
  - First the destructor of the derived class is executed
  - Then the destructor of the base class is executed
- Constructor/Destructor sequence example

```
class A {
  A() { cout << “A constructor” << endl ;}
  ~A() { cout << “A destructor” << endl ; }
}

class B : public A {
  B() { cout << “B constructor” << endl ;}
  ~B() { cout << “B destructor” << endl ; }
}

int main() {
  B b ;
  cout << endl ;
}
```

Output

```
A constructor
B constructor

B destructor
A destructor
```

Sharing information – protected access

- Inheritance preserves existing encapsulation
  - Private part of base class Employee is **not** accessible by derived class Manager

```
Manager::giveMyselfRaise() {
  _salary += 1000 ; // NOT ALLOWED: private in base class
}
```

- Sometimes useful if derived class can access part of private data of base class
  - Solution: '`protected`' -- accessible by derived class, but not by public

```
class Base {
  public:
    int a ;
  protected:
    int b ;
  private:
    int c ;
}
```

```
class Derived : public Base {
 void foo() {
   a = 3 ; // OK public
   b = 3 ; // OK protected
  }
}

Base base ;
base.a = 3 ; // OK public
base.b = 3 ; // ERROR protected
```

Better example of protected interface

```
class Employee {
public:
  Employee(const char* name, double salary) ;
  annualRaise() { setSalary(_salary*1.03) ; }
  double salary() const { return _salary ; }

protected:
  void setSalary(double newSalary) {
    if (newSalary<_salary) }
      cout << "ERROR: salary must always increase" << endl ;
    } else {
      _salary = newSalary ;
    }
  }

private:
  string _name ;
  double _salary ;
}
```

The setSalary() function is protected:

Public cannot change salary except in controlled way through public annualRaise() method

Better example of protected interface

```
class Employee {
public:
  Employee(const char* name, double
  annualRaise() { setSalary(_salary
  double salary() const { return _s

protected:
  void setSalary(double newSalary) {
    if (newSalary<_salary) }
      cout << “ERROR: salary must al      increase” << endl ;
    } else {
      _salary = newSalary ;
    }
  }

private:
  string _name ;
  double _salary ;
}
```

Managers can also get additional raise through `giveBonus()`

Access to protected `setSalary()` method allows `giveBonus()` to modify salary

```
class Manager : public Employee {
public:
   Manager(const char* name, double salary,
           list<Employee*> subs) ;

   giveBonus(double amount) {
      setSalary(salary()+amount) ;
   }
private:
  list<Employee*> _subs ;
}
```

Better example of protected interface

```
class Employee {
public:
  Employee(const char* name, double salary) ;
  annualRaise() { setSalary(_salary*1.03) ; }
  double salary() const { return _salary ; }

protected:
  void setSalary(double newSalary) {
    if (newSalary<_salary) }
      cout << "ERROR: salary must always increase" << endl ;
    } else {
      _salary = newSalary ;
    }
  }
```

Note how accessor/modifier pattern `salary()/setSalary()` is also useful for protected access

Manager is only allowed to change salary through controlled method: negative bonuses are not allowed…

```
class Manager : public Employee {
public:
   Manager(const char* name, double salary,
           list<Employee*> subs) ;

   giveBonus(double amount) {
      setSalary(salary()+amount) ;
   }
private:
  list<Employee*> _subs ;
}
```

Object Oriented Analysis & Design with Inheritance

- Principal OOAD rule for inheritance: an Is-A relation is an **extension** of an object, **not a restriction**
  - `manager` Is-An `employee` is good example of a valid Is-A relation:

    > A manager conceptually is an employee *in all respects*, but with some extra capabilities

  - *Many cases are not that simple however*

- Some other cases to consider
  - A `cat` is a `carnivore` that knows how to meow (maybe)
  - A `square` is a `rectangle` with equal sides (**no!)**
    - *'Is-A except' is a restriction, not an extension*
  - A `rectangle` is a `square` with method to change side lengths (**no!**)
    - *Code in square can make legitimate assumptions that both sides are of equal length*

Object Oriented Analysis & Design with Inheritance

- Remarkably easy to get confused
  - Particularly if somebody else inherits from your class later (and you might not even know about that)

- The Iron-Clad rule: The **L**iskov **S**ubtitution **P**rinciple
  - Original version

'If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S a subtype of T'

  - In plain English:

'An object of a subclass must behave indistinguishably from an object of the superclass when referenced as an object of the superclass'

  - Keep this in mind when you design class hierarchies using Is-A relationships

Object Oriented Analysis & Design with Inheritance

- Extension through inheritance can be quite difficult
  - ‘Family trees’ seen in text books very hard to do in real designs
- Inheritance for “extension” is non-intuitive, but for “restriction” is wrong
- Inheritance is hard to get right in advance
  - Few things are straightforward extensions
  - Often behavior needs to be overridden rather than extended
  - Design should consider entire hierarchy
- But do not despair:
  - Polymorphism offers several new features that will make OO design with inheritance easier

Advanced features of inheritance

- Multiple inheritance is also allowed
  - A class with multiple base classes

```
class Manager : public Employee, public ShareHolder {
    …
} ;
```

  - Useful in certain circumstances, but things become complicated very quickly

- Private, protected inheritance
  - Derived class does not inherit public interface of base class
  - Example declaration

```
class Stack : private List {
  …
} ;
```

  - Private inheritance does not describe a ‘Is-A’ relationship but rather a ‘Implemented-by-means-of’ relationship
  - Rarely useful
  - Rule of thumb: Code reuse through inheritance is a bad idea

Polymorphism

- Polymorphism is the ability of an object to retain its true identity even when access through a base pointer
  - This is perhaps easiest understood by looking at an example *without* polymorphism
- Example without polymorphism
  - Goal: have `name()` append `“(Manager”)` to name tag for manager
  - Solution: implement `Manager::name()` to do exactly that

```
class Manager : public Employee {
public:
  Manager(const char* name, double salary,
          vector<Employee*> subordinates) ;

  const char* name() const {
    cout << _name << “ (Manager)” << endl ;
  }

  list<Employee*> subs() const ;
private:
  list<Employee*> _subs ;
} ;
```

Example without polymorphism

- Using the improved manager class

```
Employee emp("Wouter",10000) ;
Manager mgr("Stan",20000,&emp) ;

cout << emp.name() << endl ; // Prints "Wouter"
cout << mgr.name() << endl ; // Prints "Stan (manager)"
```

- But it doesn't work in all circumstances…

```
void print(Employee& emp) {
   cout << emp.name() << endl ;
}
print(emp) ;  // Prints "Wouter"
print(mgr) ;  // Prints "Stan" – NOT WHAT WE WANTED!
```

  - **Why does this happen?**
  - Function `print()` sees `mgr` as employee, thus the compiler calls `Employee:name()` rather than `Manager::name()` ;
  - Problem profound: `name()` function call selected at compile time. No way for compiler to know that `emp` really is a Manager!

Polymorphism

- Polymorphism is the ability of an object to retain its true identity even when access through a base pointer
  - I.e. we want this:

```
Employee emp("Wouter",10000) ;
Manager mgr("Stan",20000,&emp) ;

void print(Employee& emp) {
   cout << emp.name() << endl ;
}
print(emp) ;  // Prints "Wouter"
print(mgr) ;  // Prints "Stan (Manager)"
```

- In other words: Polymorphism is the ability to treat objects of different types the same way
  - To accomplish that we will need to tell C++ compiler to look at ***run-time*** what `emp` really points to.
  - In compiler terminology this is called '***dynamic binding***' and involves the compiler doing some extra work prior executing the `emp->name()` call

Dynamic binding in C++ – keyword virtual

- The keyword ***virtual*** in a function declaration activates dynamic binding for that function
  - The example class Employee revisited

```
class Employee {
public:
  Employee(const char* name, double salary) ;
  virtual const char* name() const ;
  double salary() const ;
private:
  …
} ;
```

  - No further changes to class Manager needed

… And the broken printing example now works

```
void print(Employee& emp) {
   cout << emp.name() << endl ;
}
print(emp) ;  // Prints "Wouter"
print(mgr) ;  // Prints "Stan (Manager)" EUREKA
```

Keyword virtual – some more details

- Declaration ‘virtual’ needs only to be done in the base class
  - Repetition in derived classes is OK but not necessary

- Any member function can be virtual
  - Specified on a **member-by-member** basis

```
class Employee {
public:
  Employee(const char* name, double salary) ;
  ~Employee() ;

  virtual const char* name() const ; // VIRTUAL
  double salary() const ;            // NON-VIRTUAL

private:
  …
} ;
```

Virtual functions and overloading

- For overloaded virtual functions either all or none of the functions variants should be redefined

OK – all redefined

```
class A {
 virtual void func(int) ;
 virtual void func(float) ;
}

class B : public A {
 void func(int) ;
 void func(float) ;
}
```

OK – none redefined

```
class A {
 virtual void func(int) ;
 virtual void func(float) ;
}

class B : public A {
}
```

NOT OK – partially redefined

```
class A {
 virtual void func(int) ;
 virtual void func(float) ;
}

class B : public A {
  void func(float) ;
}
```

Virtual functions – Watch the destructor

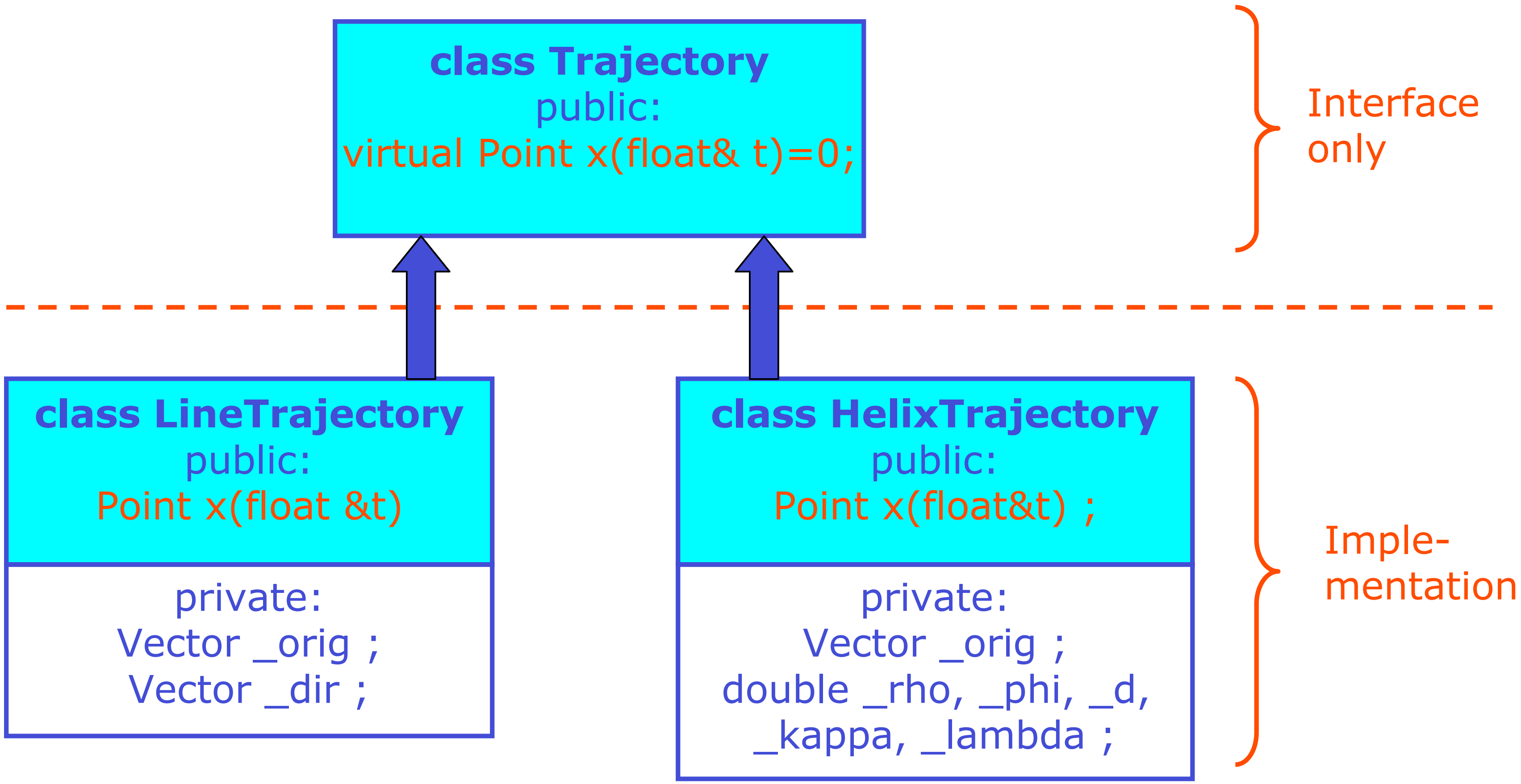
- Watch the destructor declaration if you define virtual functions
  - Example

```
Employee* emp = new Employee("Wouter",1000) ;
Manager* mgr = new Manager("Stan",20000,&emp) ;

void killTheEmployee(Employee* emp) {
   delete emp ;
}

killTheEmployee(emp) ; // OK
killTheEmployee(mgr) ; // LEGAL but WRONG!
                  // calls ~Employee() only, not ~Manager()
```

  - Any resources allocated in Manager constructor will not be released as Manager destructor is not called (just Employee destructor)
  - Solution: make the destructor virtual as well
- Lesson: if you ever delete a derived class through a base pointer your class should have a virtual destructor
  - In practice: Whenever you have any virtual function, make the destructor virtual

Abstract base classes – concept

- Virtual functions offer an important tool to OOAD – the Abstract Base Class
  - An Abstract Base Class is an interface only. It describes how an object can be used but does not offer a (full) implementation



Abstract base classes – pure virtual functions

- A class becomes an abstract base class when it has one or more pure virtual functions
  - A pure virtual function is a declaration without an implementation
  - Example

```
class Trajectory {
public:
   Trajectory() ;
   virtual ~Trajectory() ;
   virtual Point x(float& t) const = 0 ;
} ;
```

  - It is **not possible** to create an **instance** of an **abstract base class**, only of implementations of it

```
Trajectory* t1 = new Trajectory(…) ;    // ERROR abstract class
Trajectory* t2 = new LineTrajectory(…); // OK
Trajectory* t3 = new HelixTrajectory(…);// OK
```

Abstract base classes and design

- Abstract base classes are a way to express common properties and behavior without implementation
  - Especially useful if there are multiple implementation of a commons interface possible
  - Example: a straight line '**is a**' trajectory,
    but a helix also '**is a**' trajectory

- Enables you to write code at a higher level abstraction
  - For example, you don't need to know how trajectory is parameterized, just how to get its position at a give flight time.
  - Powered by polymorphism

- Simplifies extended/augmenting existing code
  - Example: can write new class `SegmentedTrajectory`. Existing code dealing with trajectories can use new class without modifications (or even recompilation!)

Abstract Base classes – Example

- Example on how to use abstract base classes

```
void processTrack(Trajectory& track) ;

int main() {
  // Allocate array of trajectory pointers
  Trajectory* tracks[3] ;

  // Fill array of trajectory pointers
  tracks[0] = new LineTrajectory(…) ;
  tracks[1] = new HelixTrajectory(…) ;
  tracks[2] = new HelixTrajectory(…) ;

  for (int i=0 ; i<3 ; i++) {
    processTrack(*tracks[i]) ;
  }
}

void processTrack(Trajectory& track) {
   cout << "position at flight length 0 is "
        << track.pos(0) << endl ;
}
```

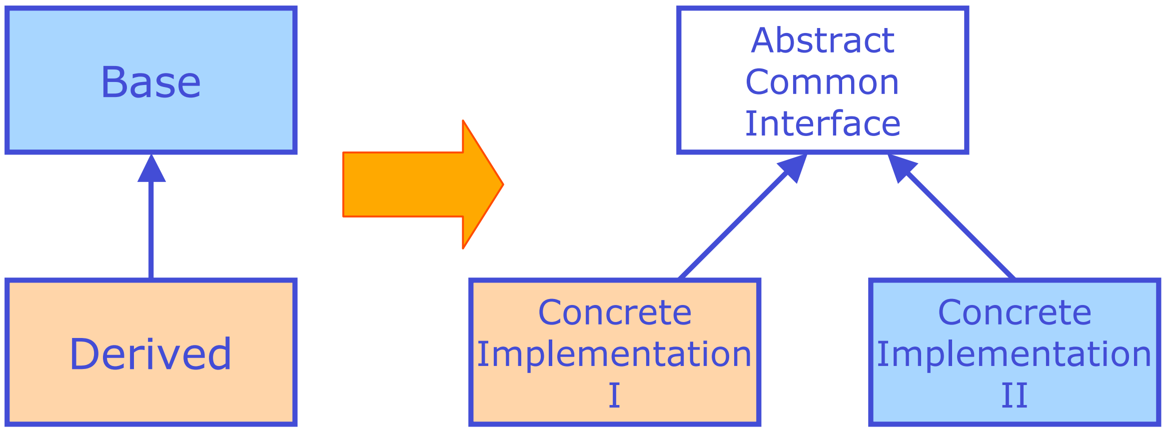
Use Trajectory interface to manipulate track without knowing the exact class you're dealing with (HelixTrajectory or LineTrajectory)

The power of abstract base classes

- You can even reuse existing *compiled* code with new implementations of abstract base classes
- Example of reusing compiled code with a new class
  - First iteration – no magnetic field
    1. Write abstract class `Trajectory`
    2. Write implementation `LineTrajectory`
    3. Write algorithm class `TrackPointPOCA` to find closest point of approach between given cluster position and trajectory using `Trajectory` interface
  - Second iteration – extend functionality to curved tracks in magnetic field
    1. Write implementation `HelixTrajectory`, compile `HelixTrajectory` code
    2. Link `HelixTrajectory` code with existing compiled code into new executable
    3. Your executable can use the newly defined `HelixTrajectory` objects without further modification
- Higher level code `TrackPointPOCA` transparent to future code changes!

Object Oriented Analysis and Design and Polymorphism

- Design of class hierarchies can be much simplified if only abstract base classes are used
  - In plain inheritance derived class forcibly inherits full specifications of base type
  - Two classes that inherit from a common abstract base class can share any subset of their common functionality



Polymorphic objects and storage

- Polymorphic inheritance simplifies many aspects of object use and design – but there are still some areas where you still need to pay attention:

- Storage of polymorphic object collections
  - Reason: when you start allocating memory the true identity of the matters. you need to know exactly how large it is after all…
  - Storage constructions that assume uniform size of objects also no longer work – Use of arrays, STL container classes not possible

- Cloning of polymorphic object collections
  - Reason: you want to clone the implementation class not the interface class so you must know the true type
  - Ordinarily virtual functions solves such problems, however there is no such thing as a virtual copy constructor…

- Will look into this in a bit more detail in the next sides…

Collections of Polymorphic objects – storage

- Dealing with storage
  - Naïve attempt to make STL list of trajectories

```
LineTrajectory track1(…) ;
HelixTrajectory track2(…) ;

list<Trajectory> trackList ; // ERROR
```

  - Why Error: list<X> calls default constructor for X, but can not instantiate X if X is an abstract classes such as Trajectory
  - Solution: make a ***collection of pointers***

```
Trajectory* track1 = new LineTrajectory(…) ;
Trajectory* track2 = new HelixTrajectory (…) ;

list<Trajectory*> trackList ; // OK
trackList.push_back(&track1) ;
trackList.push_back(&track2) ;
```

Collections of Polymorphic objects – storage

- But remember ownership semantics
  - STL container will delete pointers to objects, but not objects themselves
  - In other words: **deleting `trackList` does NOT delete the tracks!**

- Technical Solution
  - Write a new container class, or inherit it from a STL container class that takes ownership of objects pointed to.
  - NB: This is not so easy – think about what happens if replace element in container: does removed element automatically get deleted on the spot?

- Bookkeeping Solution
  - Document clearly in function that creates `trackList` that contents of `tracklist` is owned by caller in addition to list itself
  - More prone to mistakes

Collections of polymorphic objects – copying

- Copying a polymorphic collection also has its issues

```
list<Trajectory*> trackList ;
list<Trajectory*> clonedTrackList ;

list<Trajectory*>::iterator iter ;
for(iter=trackList.begin() ; iter!=trackList.emd() ; ++iter) {
  Trajectory* track = *iter ;

  newTrack = new Trajectory(*track) // NOPE – attempt to
                                    // instantiate abstract class
  cloneTrackList.push_back(newTrack) ;
}
```

- Solution: make your own 'virtual copy constructor'
  - Add a pure virtual `clone()` function to your abstract base class

```
class Trajectory {
public:
   Trajectory() ;
   virtual ~Trajectory() ;
   virtual Trajectory* clone() const = 0 ;
   virtual Point x(float& t) const = 0 ;
} ;
```

The virtual copy constructor

- Implementing the `clone()` function

```
class LineTrajectory : public Trajectory {
   LineTrajectory(…) ;
   LineTrajectory(const LineTrajectory& other) ;
   virtual ~LineTrajectory() ;

   // 'virtual copy constructor'
   virtual Trajectory* clone() const {
       return new LineTrajectory(*this) ; calls copy ctor
   }
}
```

- Revisiting the collection copy example

```
list<Trajectory*>::iterator iter ;
for(iter=tl.begin() ; iter!=tl.end() ; ++iter) {
  Trajectory* track = *iter ;
  newTrack = track->clone() ;
  clonedTrackList.push_back(newTrack) ;
}
```

   - `clone()` returns a `Trajectory*` pointer to a `LineTrajectory` for track1
   - `clone()` returns a `Trajectory*` pointer to a `HelixTrajectory` for track 2

Run-time type identification

- Sometimes you need to cheat…
  - Example: The preceding example of cloning a list of tracks
  - Proper solution: add `virtual clone()` function
  - But what if (for whatever reason) we cannot touch the base class?
    - For example: it is designed by somebody else that doesn't want you change it, or it is part of a commercial library for which you don't have the source code
  - Can you still tell what the true type is given a base class pointer?

- Solution: the `dynamic_cast<>` operator
  - Returns valid pointer if you guessed right, null otherwise

```
Trajectory* track ;
LineTrajectory* lineTrack =
dynamic_cast<LineTrajectory*>(track) ;

if (lineTrack != 0) {
  cout << "track was a LineTrajectory" << endl ;
} else {
  cout << "track was something else" << endl ;
}
```

Run time type identification

- Solution to `trackList` clone problem

```
list<Trajectory*>::iterator iter ;
for(iter=tl.begin() ; iter!=tl.end() ; ++iter) {
  Trajectory* track = *iter ;

  LineTrajectory* line = dynamic_cast<LineTrajectory*> track ;
  if (line) {
    newTrack = new LineTrajectory(*line) ;
    continue ;
  }

 HelixTrajectory* helix = dynamic_cast<HelixTrajectory*> track;
  if (helix) {
    newTrack = new HelixTrajectory(*helix) ;
    continue ;
  }

 cout << "ERROR: track is neither helix nor line"
}
```

  - *Obviously ugly, maintenance prone, incomplete*
  - **Use `dynamic_cast<>` as last resort only!**

C++ competes with your government

- Flip side of polymorphic inheritance – performance

- Inheritance can be taxed!
  - In C++ you incur a performance overhead if you use virtual functions instead of regular (statically bound) functions
  - Reason: every time you call a virtual function the C++ compiler inserts code that identifies the true identity of the object and decided based on that information what function to call
  - Overhead *only* applies to virtual functions. Regular function in a class with other virtual functions do not incur the overhead

- Use virtual functions judiciously
  - Don't make every function in your class virtual
  - Overhead is not always waste of time. If alternative is figuring out the true identity of the object yourself the lookup step is intrinsic to your algorithms.

9 Exception handling

Introduction to Exception handling

- Exception handling = handling of anomalous events outside regular chain of execution
  - Purpose: Handle error conditions that cannot be dealt with locally

- Your traditional options for dealing with an error
  1. Terminate the program
     - *Not acceptable for embedded program, e.g high voltage controller*
  2. Return a value representing an error
     - *Often return type has no 'reserved values' that can be used as error flag*
  3. Return a legal value but set a global error flag
     - *You make the tacit assumption that somebody will check the flag*
  4. Call a user supplied error function
     - *Just passing the buck – user function also has no good option to deal with problem*

Throwing and catching

- Simple example of C++ exception handling
  - Exception is ‘thrown’ in case of run-time problem
  - If exception is not caught (as in example below) program terminates

```
int main() {
   double x(-3) ;
   double y = calc(x) ;
}


double calc(double x) {
  if (x<0) {
    // run-time error condition
    throw x ;
  }
  return sqrt(x) ;
}
```

Exceptions and automatic variables

- How is throwing an exception better than calling abort()?
  - If exception is thrown destructor is called for all automatic variables up to point where exception is handled (in this case up to the end of `main()`)
  - In example below, buffers of output file `ofs` flushed, and file closes properly prior to program termination

```
int main() {
   SomeClass obj ;
   double x(-3) ;
   double y = calc(x) ;
}


double calc(double x) {
  ofstream ofs ;
  if (x<0) {
    // run-time error condition
    throw x ;
  }
  return sqrt(x) ;
}
```

time line

```
1)  throw f
2)  ofs destructor
    (closes file)
3)  obj destructor
4)  exit
```

Catching exceptions

- You can also deal explicitly with exception 'catching' the exception
  - You can pass information to the handler via the thrown object

```
int main() {
   double x(-3) ;
   try {
     double y = calc(x) ;
   }
   catch (double x) {
     cout << "oops, sqrt of "
          << "negative number:"
          << x << endl ;
   }
}

double calc(double x) {
  if (x<0) {
    // run-time error condition
    throw x ;
  }
  return sqrt(x) ;
}
```

Exceptions are caught in this {} block

Exceptions handled in this block

Display details on error using information passed by exception
(value of negative number in this case)

Catching deep exceptions

- Exceptions are also caught by `try{},catch{}` if they occur deeply inside nested function calls

```
int main() {
   double x(-3) ;
   try {
     double y = wrapper(x) ;
   }
   catch (float x) {
     cout << “
          << ”
          << e
   }
}
```

```
double wrapper(double x) {
  // do some other stuff
  return calc(x) - 5 ;
}
```

```
double calc(double x) {
  if (x<0) {
    // run-time error condition
    throw f ;
  }
  return sqrt(x) ;
}
```

Solving the problem

- You can try to solve the problem in the catch block
  - If you fail, call `throw` inside the catch block to indicate that you give up

```
int main() {
   double x(-3) ;
   try {
     int* array = allocate(10000) ;
   }
   catch (int size) {
     cout << "error allocating array of size "
          << size << endl ;

     // do some housekeeping

     if (problem_solved) {
       array = allocate(size) ;
     } else {
       throw ; // give up handling error here
     }
   }
}
```

If allocate throws an exception again it will not be caught by surrounding catch block

Solving the problem in steps

- A chain of error handlers
  - A rethrow allows a higher level error handler to deal with the problem if you can't

```
int main() {
   double x(-3) ;
   try {
     double y = wrapper(x) ;
   }
   catch (float x) {
     // second level error handling
   }
}

double wrapper(double x) {
  // do some other stuff
  try {
      calc(x) ;
  }
  catch (float x) {
     // first resort error handling
     if (!problem_solved) {
        throw ; // forward
     }
  }
}

double calc(double x) {
   throw f ;
}
```

There are exceptions and exceptions

- Usually more than one kind of error can occur
  - Give each error its own type of exception. Each type that is thrown can be assigned a separate catch handler:

```
int main() {
   double x(-3) ;
   try {
     double y = calc(x) ;
   }
   catch (int x) {
     cout << "oops, sqrt of "
          << "negative integer number"
          << endl ;
   }
   catch (float x) {
     cout << "oops, sqrt of "
          << "negative floating point number"
          << endl ;
   }
}
```

The catchall

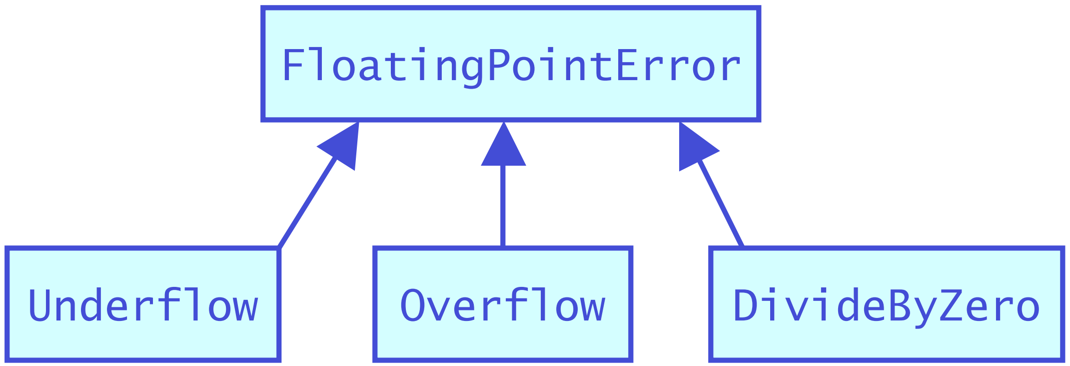
- When you specify multiple exception handler they are tried in order
  - *The catch(…) handler catches any exception*. Useful to put last in chain of handlers

```
int main() {
   double x(-3) ;
   try {
     double y = calc(x) ;
   }
   catch (int x) {
     // deal with int exception
   }
   catch (float x) {
     // deal with float exception
   }
   catch (…) {
     // deal with any other exception
   }
}
```

Exceptions and objects – hierarchy

- So far example have thrown floats and ints as exceptions
- What else can we throw?
  - Actually, *anything*, including objects!

- Throwing objects is particularly nice for several reason
  1. Error often have a hierarchy, just like objects can have
  2. Objects can carry more information, error status, context etc …
  - Example of exception hierarchy



Throwing objects

- Example of hierarchical error handling with classes
  - The hierarchy of throwable classes

```
class FloatingPointError {
  FloatingPointError(float val) : value(val) {}
  public:
  float value ;
}

class UnderflowError : public FloatingPointError {
  UnderflowError(float val) : FloatingPointError(val) {}
}

class OverflowError : public FloatingPointError {
  OverflowError(float val) : FloatingPointError(val) {}
}

class DivZeroError : public FloatingPointError {
  DivZeroError(float val) : FloatingPointError(val) {}
}
```

Catching objects

- Hierarchical handling of floating point errors
  - Note that if we omit the `DivZeroError` handler that exception is still caught (but then by the `FloatingPointError` handler)

```
void mathRoutine() {
   try {
      doTheMath() ;
   }

   // Catches divide by zero errors specifically
   catch(DivZeroError zde)  {
      cout << "You divided " << zde.value
           << " by zero!" << endl ;
  }

  // Catches all other types of floating point errors
  catch(FloatingPointError fpe) {
     cout << "A generic floating point error occurred,"
          << " value = " << fpe.value << endl ;
  }
}
```

Throwing polymorphic objects

- We can simplify error further handling by throwing polymorphic objects

```
class FloatingPointError {
  FloatingPointError(float val) : value(val) {}
  virtual const char* what() { return "FloatingPointError" ; }
  public:
  float value ;
}

class UnderflowError : public FloatingPointError {
  UnderflowError(float val) : FloatingPointError(val) {}
  const char* what() { return "UnderflowError" ; }
}

class OverflowError : public FloatingPointError {
  OverflowError(float val) : FloatingPointError(val) {}
   const char* what() { return "OverflowError" ; }
}

class DivZeroError : public FloatingPointError {
  DivZeroError(float val) : FloatingPointError(val) {}
   const char* what() { return "DivZeroError" ; }
}
```

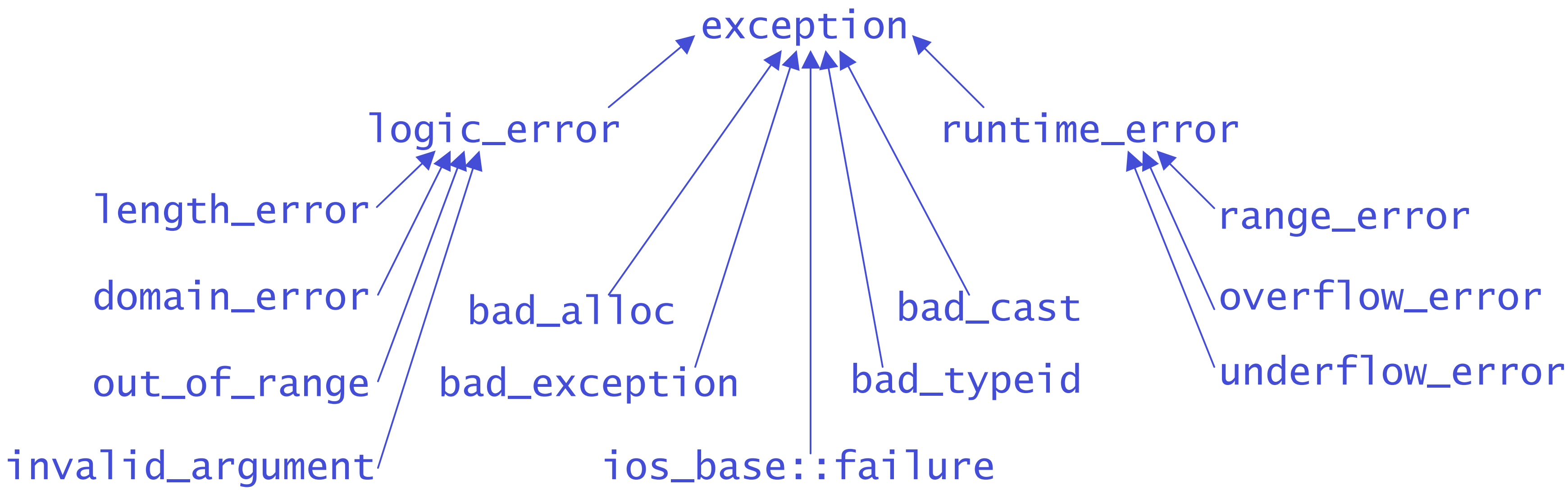
Catching polymorphic objects

- Handling of all `FloatingPointErrors` in a single handler
  - Of course only works if action for all type of errors is the same (or implemented as virtual function in exception class)
  - If structurally different error handling is needed for one particular type of `FloatingPointError` you can still insert separate handler

```
void mathRoutine() {
   try {
      doTheMath() ;
   }

  // Catches all types of floating point errors
  catch(FloatingPointError fpe) {
     cout << fpe.what() << endl ;
  }
}
```

Standard Library exception classes

- The Standard Library includes a hierarchy of objects to be thrown as exception objects
  - Base class ‘exception’ in header file <exception>
- Hierarchy of standard exceptions
  - Member function what() returns error message



  - Note that catching class exception does not guarantee catching all exceptions, people are free to start their own hierarchy

10 Where to go from here

Recommended reading

- The C++ syntax in alls its detail
  - The C++ programming Language , 3rd edition (Bjarne Stroustrup)

- How to program good C++
  - Catching those hidden performance issues
  - Good coding practices etc etc
  - 'Effective C++' & 'More Effective C++' (Scott Meyers)

- A book with a lot of practical examples
  - 'Accelerated C++' (Andrew Koenig & Barbara Moe)

- Software design
  - 'Design Patterns' (Gamma et al.) [not for beginners]

Web resources

- Frequently Asked Questions on C++ (and answers)
  - `http://www.parashift.com/c++-faq-lite/`

- A reference guide to the Standard Template Library
  - `http://www.sgi.com/tech/stl/`