Introduction to C++
and Object Oriented Programming

Wouter Verkerke (NIKHEF)

v60 – Edition for 2018 Master Course

0 Introduction & Overview

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 has allowed us to tackle more and more complex problems
- As a consequence software has also grown more powerful and complex
  - For example Microsoft Windows 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

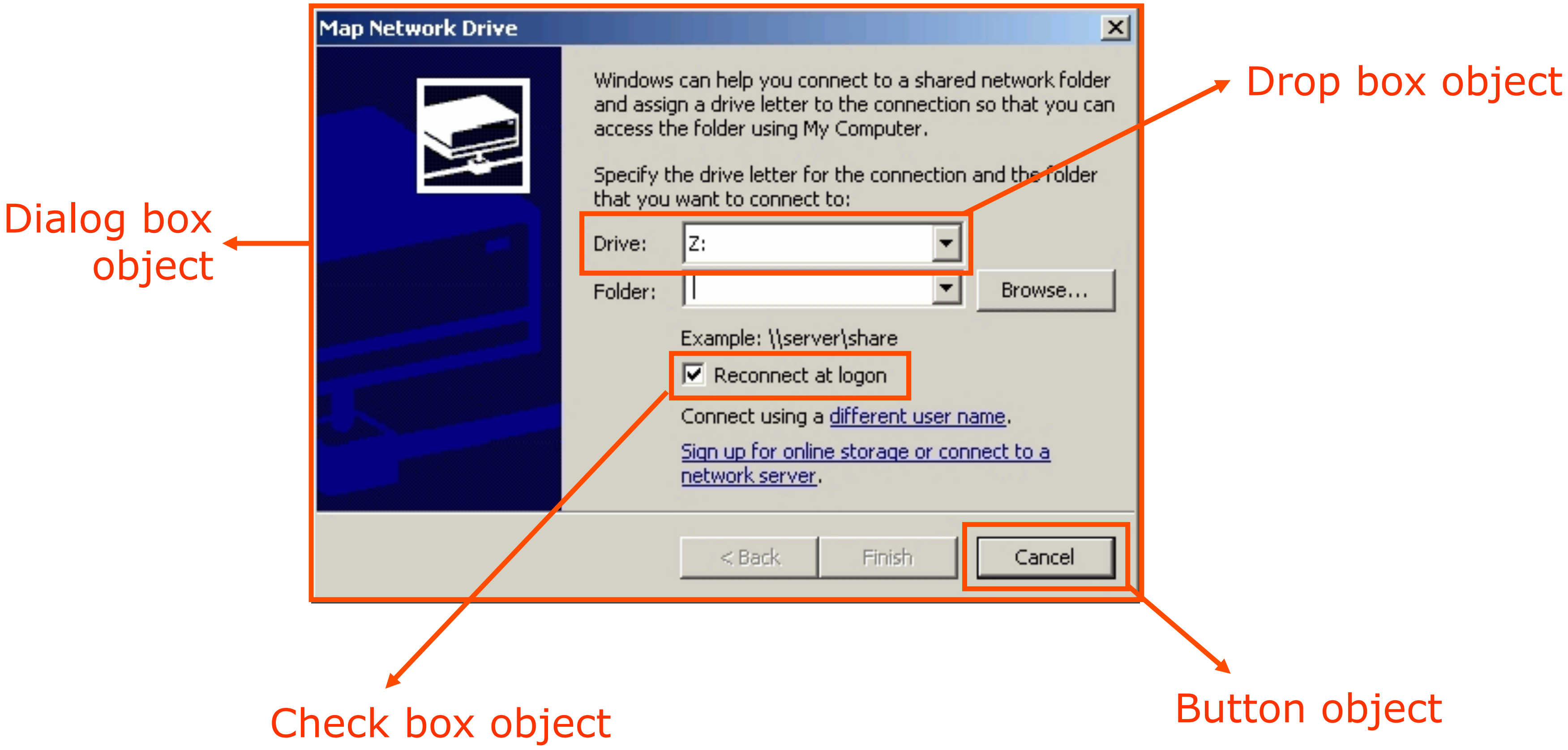
- Key to successfully coding complex systems is break down code into smaller modules and minimize the dependencies between these modules

- 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



Map Network Drive

Windows can help you connect to a shared network folder and assign a drive letter to the connection so that you can access the folder using My Computer.

Specify the drive letter for the connection and the folder that you want to connect to:

Drive: Z:

Folder:

Example: \\server\share

☑ Reconnect at logon

Connect using a different user name.

Sign up for online storage or connect to a network server.

Drop box object

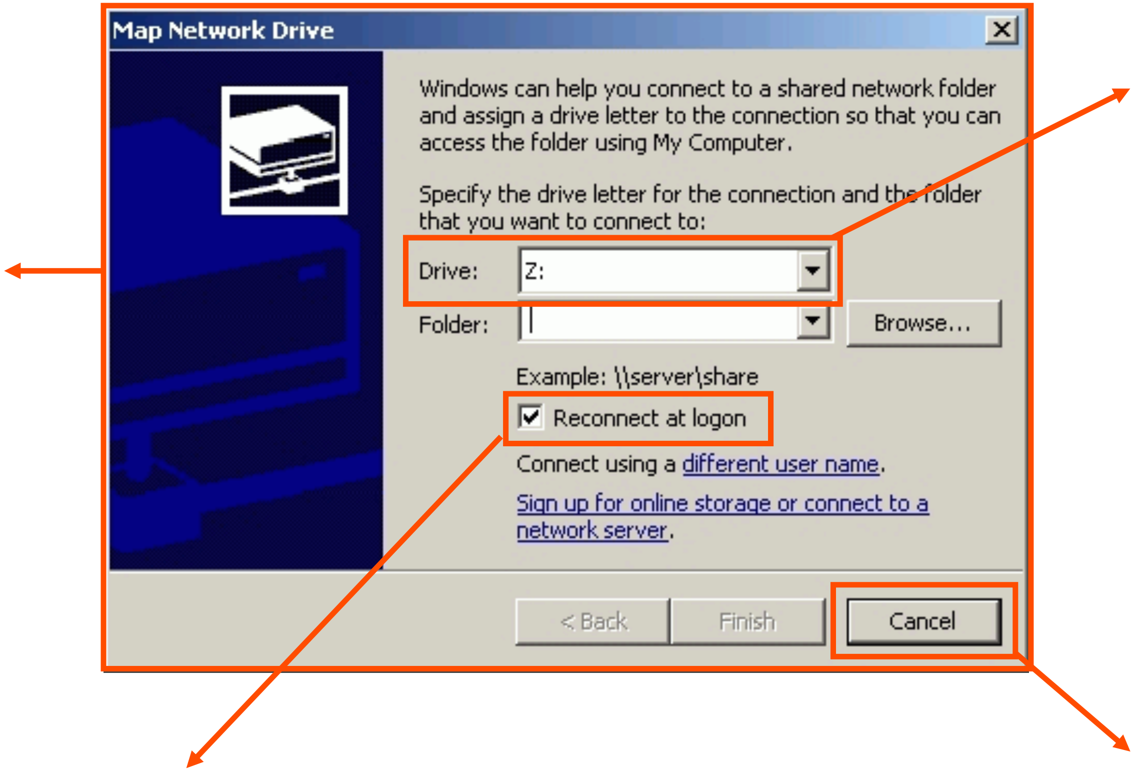
Dialog box object

Check box object

Button object

What are objects

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



Map Network Drive

Windows can help you connect to a shared network folder and assign a drive letter to the connection so that you can access the folder using My Computer.

Specify the drive letter for the connection and the folder that you want to connect to:

Drive: Z:

Folder:

Example: \\server\share

☑ Reconnect at logon

Connect using a different user name.

Sign up for online storage or connect to a network server.

Dialog box object

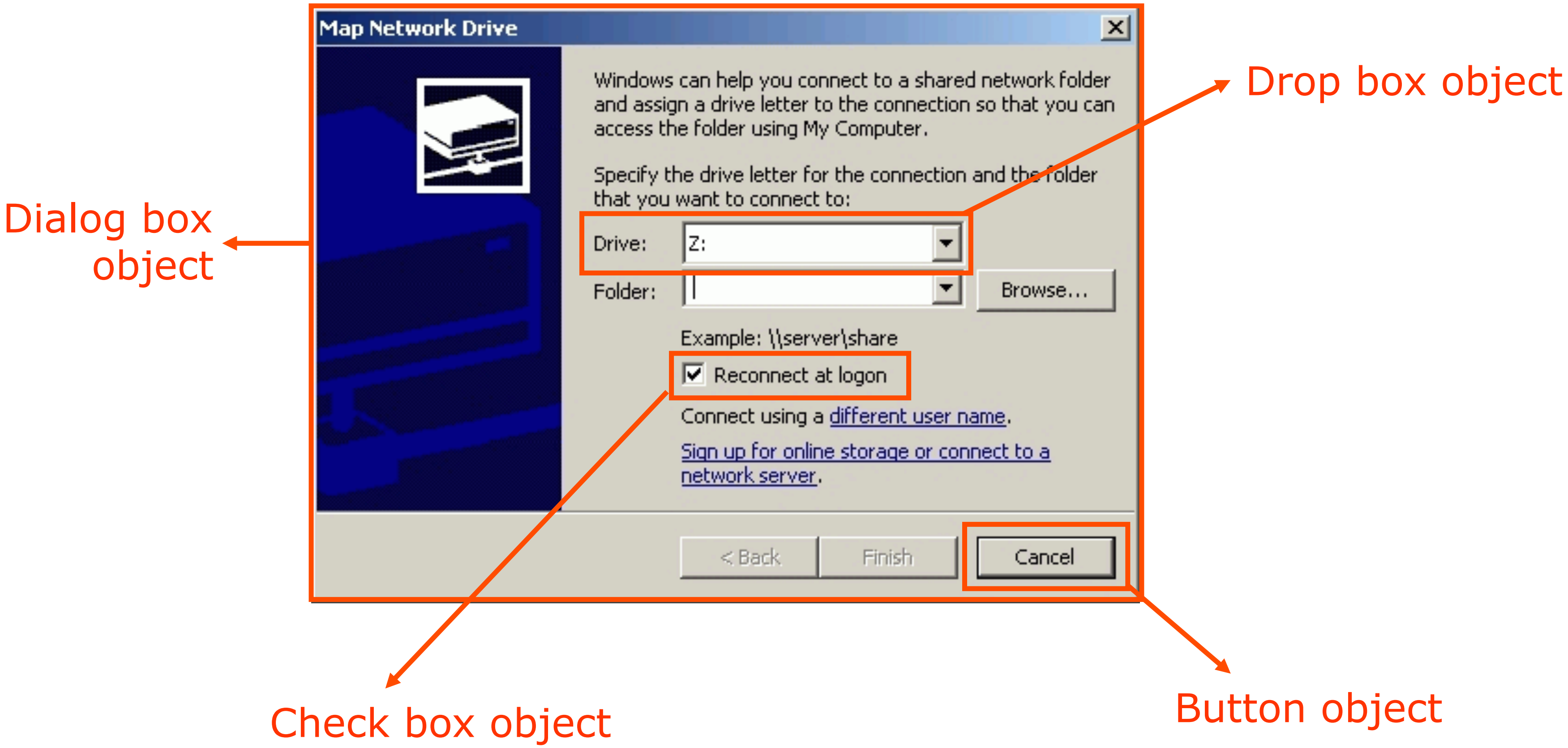
Drop box object

Check box object

Button object

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)



Map Network Drive

Windows can help you connect to a shared network folder and assign a drive letter to the connection so that you can access the folder using My Computer.

Specify the drive letter for the connection and the folder that you want to connect to:

Drive: Z:

Folder:

Example: \\server\share

☑ Reconnect at logon

Connect using a different user name.

Sign up for online storage or connect to a network server.

Dialog box object

Drop box object

Check box object

Button object

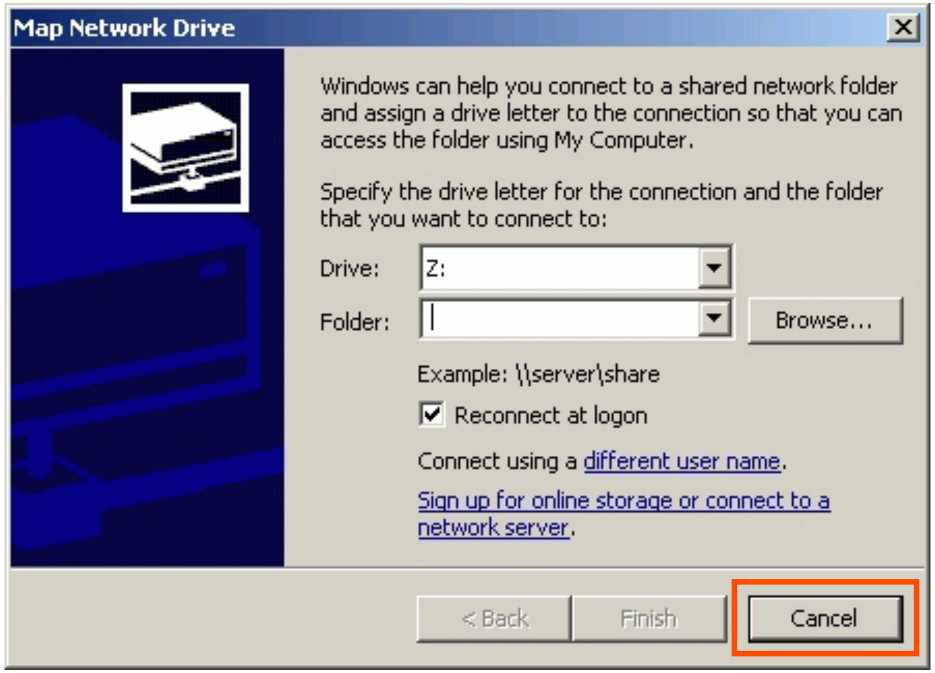
Benefits of Object-Oriented programming

- Benefits of Object-oriented programming
  - Reuse of existing code – objects can represent generic problems
  - Improved maintainability – objects are more self contained than ‘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*



Windows can help you connect to a shared network folder and assign a drive letter to the connection so that you can access the folder using My Computer.

Specify the drive letter for the connection and the folder that you want to connect to:

Drive: Z:

Folder:

Example: \\server\share

☑ Reconnect at logon

Connect using a different user name.

Sign up for online storage or connect to a network server.

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

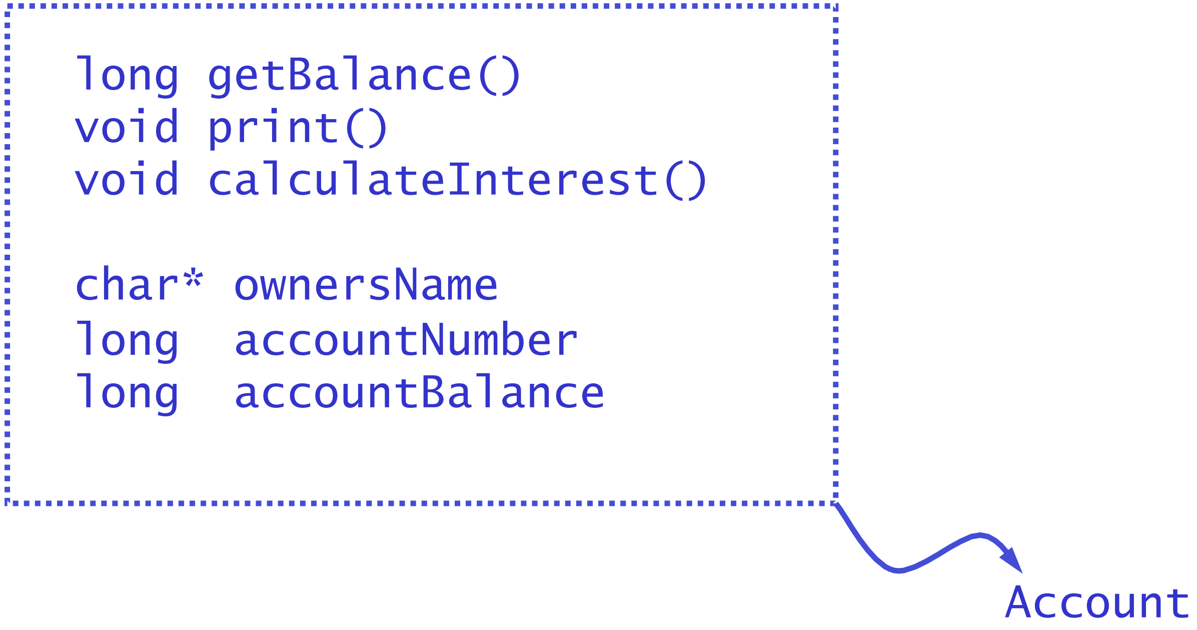
- Abstraction is achieved through

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

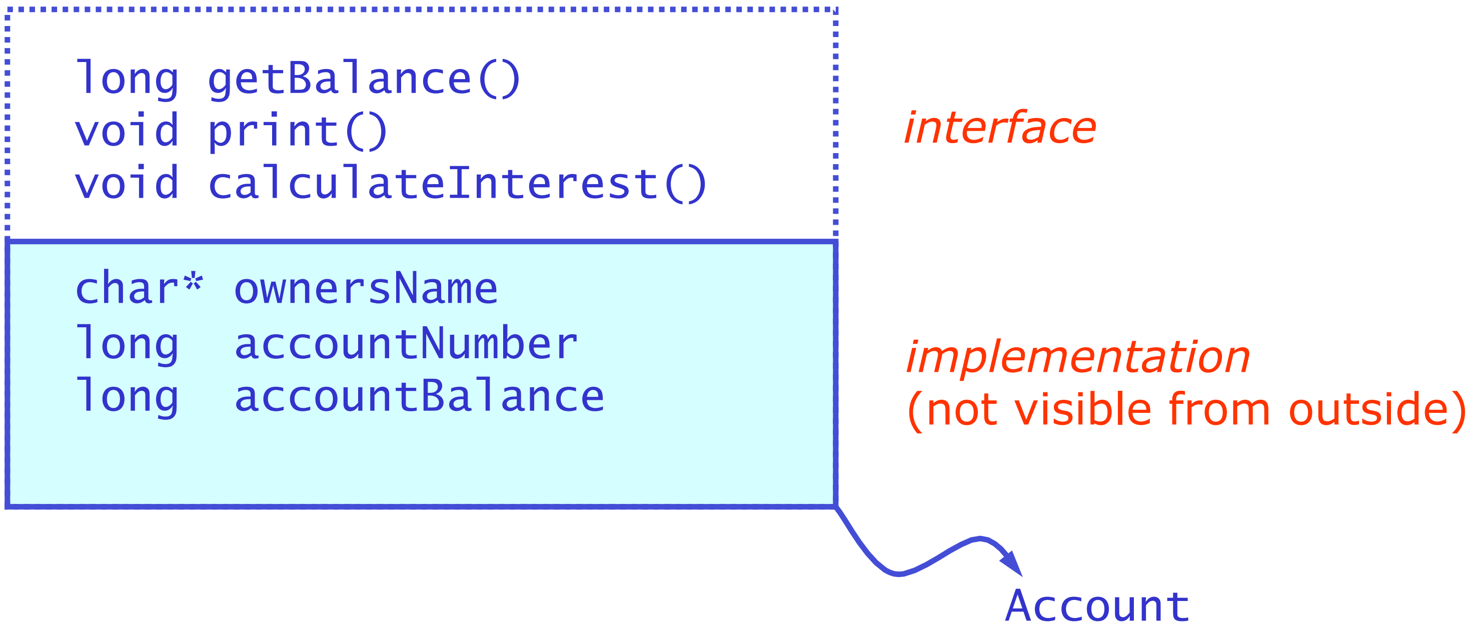


```
long getBalance()
void print()
void calculateInterest()

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

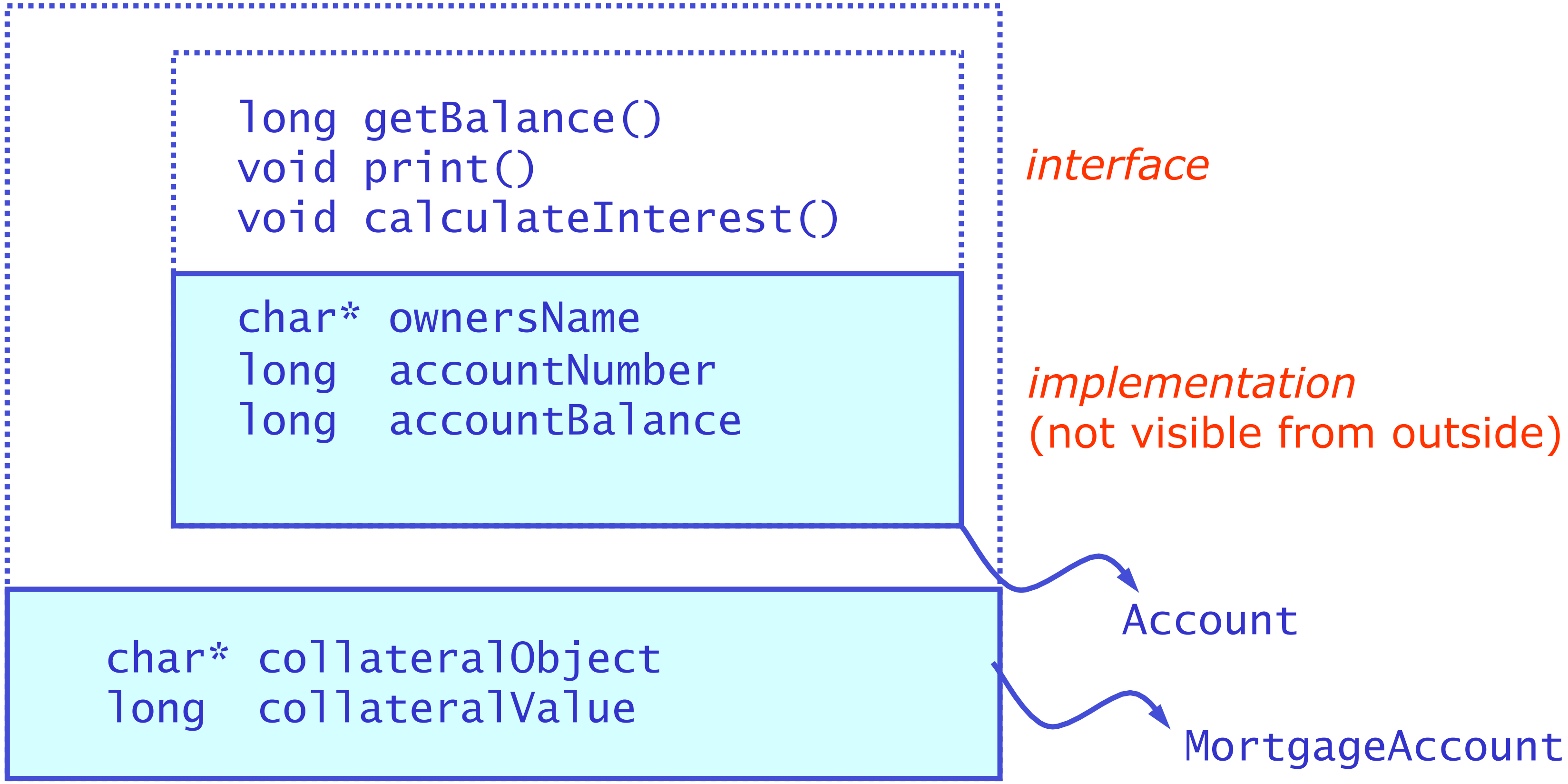
```
char* ownersName
long  accountNumber
long  accountBalance
```

implementation
(not visible from outside)

Account

Inheritance

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



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

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

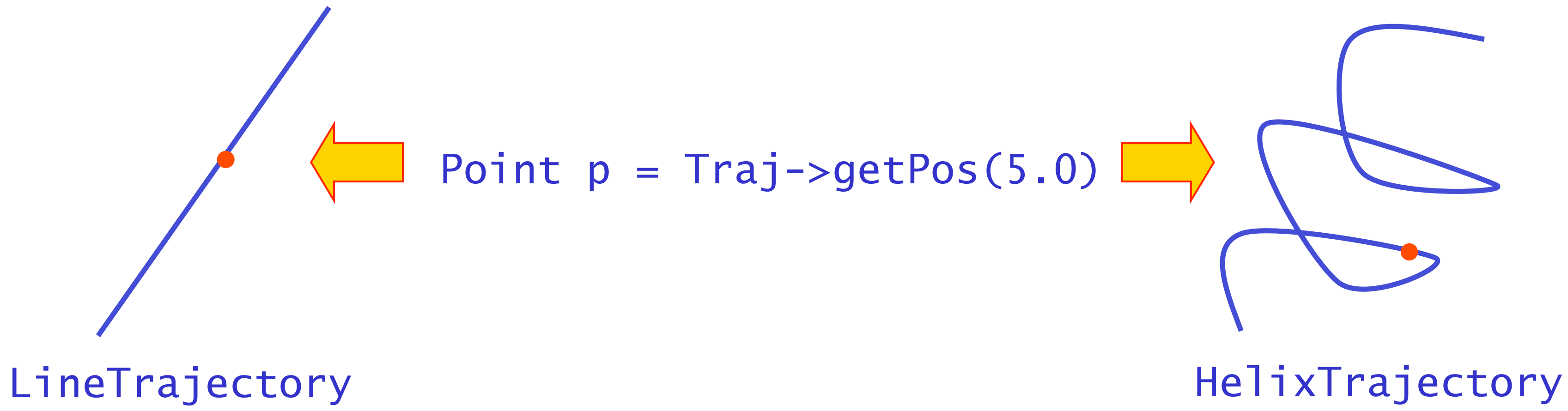
```
char* collateralObject
long  collateralValue
```

interface

implementation
(not visible from outside)

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



```
Point p = Traj->getPos(5.0)
```

LineTrajectory

HelixTrajectory

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

Versions of C++

- C++ is a ‘living language’ that evolves over time.
- This course is largely based on the 2003 standard of C++
- LHC experiments are now largely adopting C++ compilers that implement the 2011 standard of C++, which brings useful new features
  - E.g. Auto types, range-based for loops, lambdas, constructor delegation, tuples, hash tables and pointer memory management
  - I will cover a subset of these C++2011 features in this course, and explicitly point out the features that are only available in C++2011
- For the GNU compilers (gcc/g++) some of the C++2011 features are implement starting in version 4.4, with almost all features implemented in 4.7
  - In gcc 4.[3456] must add flag ‘-std=c++0x’ to activate
  - In gcc 4.[78] must add flag ‘-std=c++11’ to activate

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++

- Lets 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++

- Lets 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++

- Lets 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 to 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++ programs 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 world” in C++

- Lets start with a very simple C++ program

```
// my first program
#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 world” in C++

- Lets 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 passes 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++ expression

   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
}
```

Auto declaration type **(C++ 2011)**

- In C++ 2011, you can also omit an explicit type in declarations of objects that are immediately initialized
- In these cases the type is deduced from the initializer

```
auto j = 16 ;   // j is integer
auto j = 2.3 ;  // j is double
auto j = true ;  // j is bool
```

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

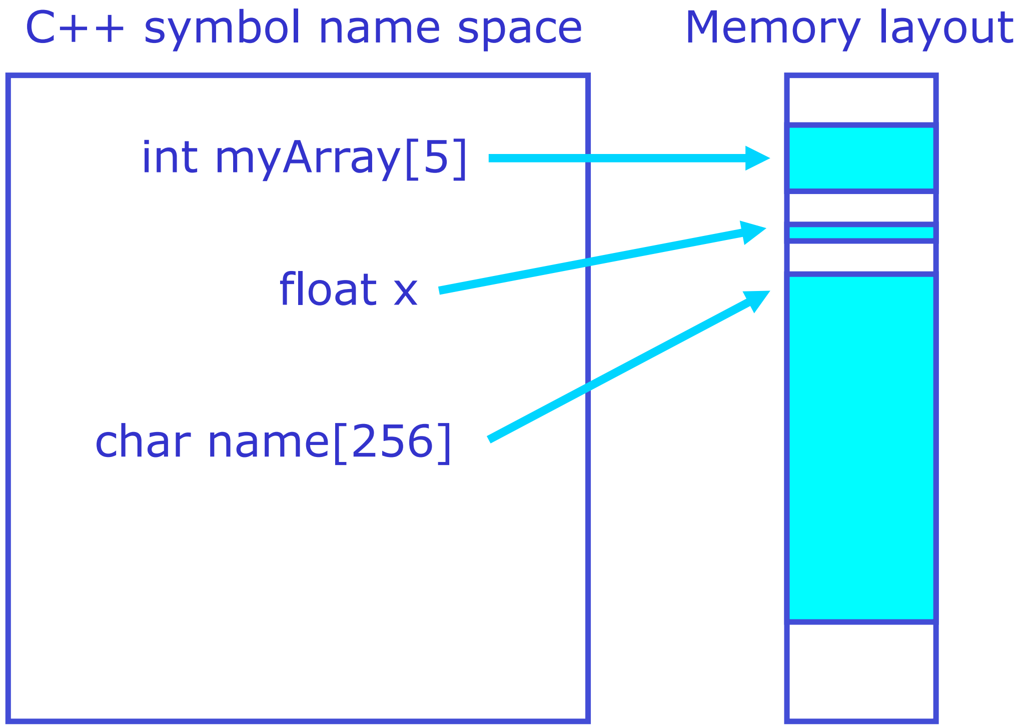
```
float x[3] ;     // OK

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

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

  - *First element's index 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

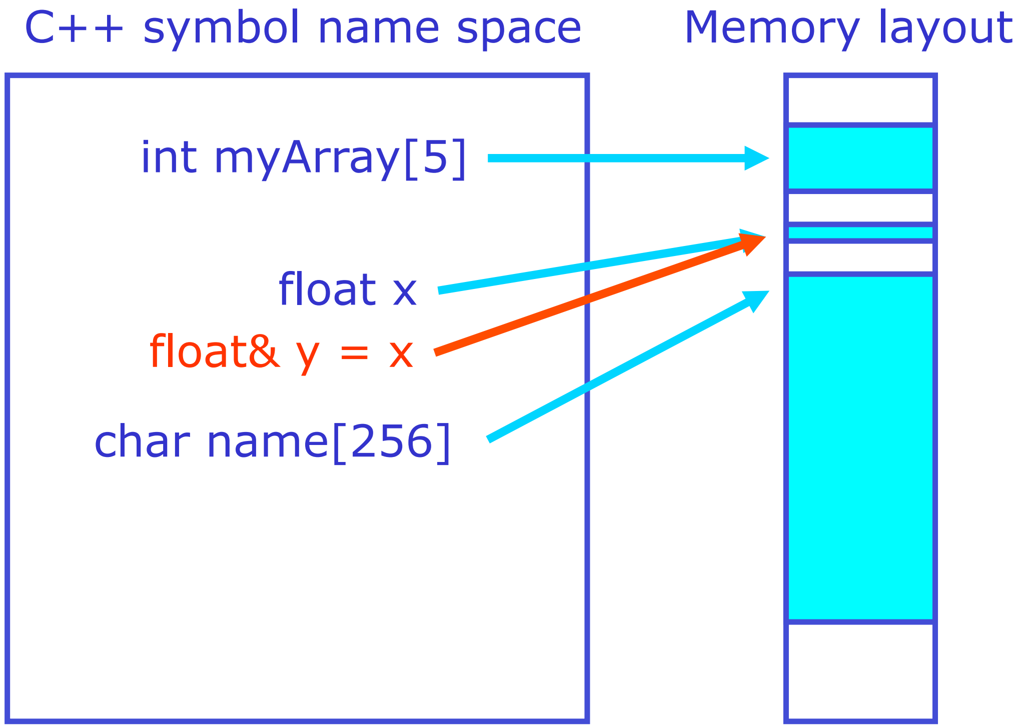
- C++ allows to create ‘alias names’, a different symbolic name referencing an already allocated data object
  - Syntax: ‘`Type& name = othername`’
  - References do not necessarily allocate memory
- Example

```
int x ;       // Allocation of memory for int
              // and declaration of name ‘x’
int& y = x ;  // Declaration of alias name ‘y’
              // for memory referenced 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

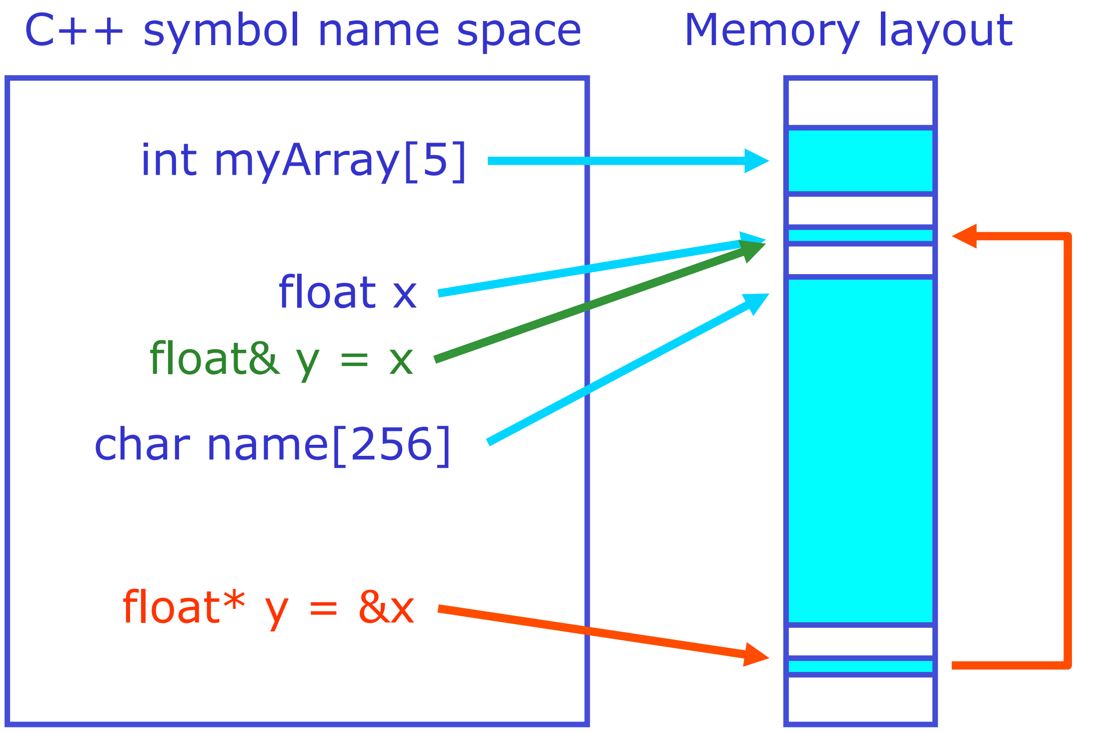
References

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



Pointers

- Pointer 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 << endl ;// 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, object 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 to 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)**

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)

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

  if (x>0) {
    int i = 0 ;
    // code
  } else {
    // code
  }
}
```

Memory for ‘int i’ allocated

Memory for ‘int i’ released

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

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 unknown size 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 object
Type* ptr = new Type ;
Type* ptr = new Type(initValue) ;

// Arrays of objects
Type* ptr = new Type[size] ;
Type* ptr = new Type[size1][size2]…[sizeN] ;
```

Releasing dynamic memory allocation

- Operator delete releases dynamic memory previously allocated with new

```
// Single object
delete ptr ;

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

  - **Be sure to use delete[] for allocated arrays**. A mismatch will result in an incomplete memory release
  - **The delete operator only deletes memory that the pointer points to, not pointer itself**
  - **Every call to new must be matched with a call to a delete**

- 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 its 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 after use

- 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 ;
   }
   return array;
}

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

  delete[] array ;
}
```

2 Files and Functions

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
}
```

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) ;

int 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 ;
}
// outputs: “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, because 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) ;

int 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 ;
}
// outputs: “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 objects)
  - 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) ;

int 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 ;
}

// outputs: “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 – 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, const 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) ;

int 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=-3.0 ;

  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=-3.0 ;

  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

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

    ```
    #ifndef 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

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`

Debugging tips – Crashes etc...

- Your program crashes – How do you analyze this
  - Recompile your program with the '-g' flag
    (i.e. `g++ -g –o blah blah.c`).
    - This will preserve source code line-number information in the executable
  - Rerun your program in the debugger:

    ```
    unix> gdb blah
    gdb> run <command line args for blah, if any, go here>
    ```

    (wait for crash)

    ```
    gdb> where
    ```

    (shows line of code where crash occurred)

    ```
    gdb> quit
    ```

    (exits the debugger)

Debugging tips – Memory leaks, corruption etc

- You want to check that no memory leaks occur, no memory corruption occurs (e.g. writing beyond boundaries of arrays etc...)
  - Recompile your program with the '-g' flag (i.e. `g++ -g -o blah blah.c`).
    - This will preserve source code line-number information in the executable
  - Rerun your problem with valgrind

    ```
    unix> valgrind blah
    ```

  - If memory corruption occurs, ERRORs will be printed in report (along with line numbers in code)
  - If memory leakage occurs, only total amount leaked is shown. To show report with details (where memory was allocated that was not deleted), rerun

    ```
    unix> valgrind --leak-check=full blah
    ```

3 Class Basics

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

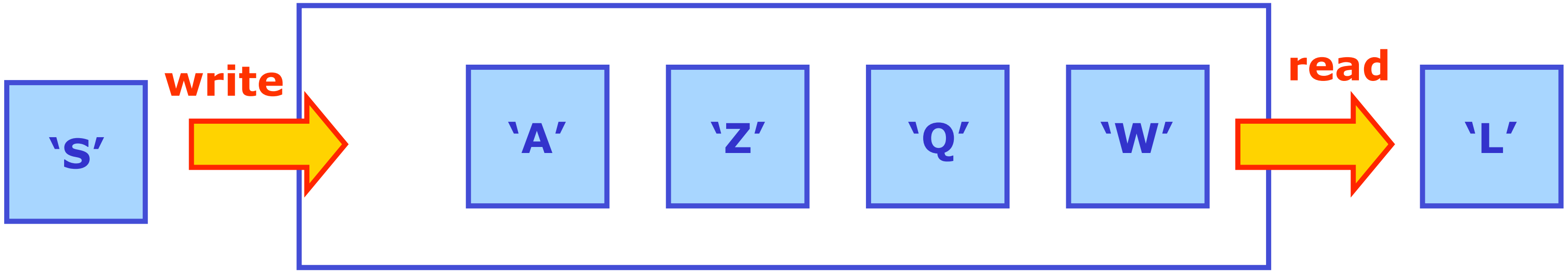
- Example

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

Evolving code design through use of C++ classes

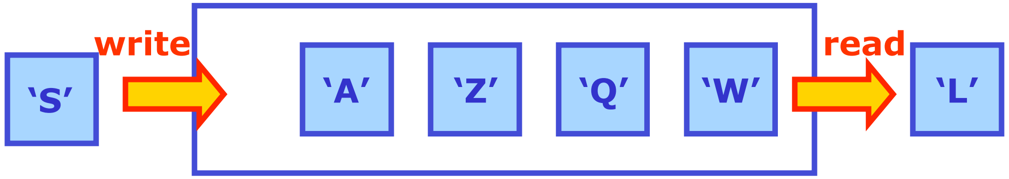
- Illustration of utility of C++ classes – Designing and building a FIFO queue
  - FIFO = '**F**irst **I**n **F**irst **O**ut'

- 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

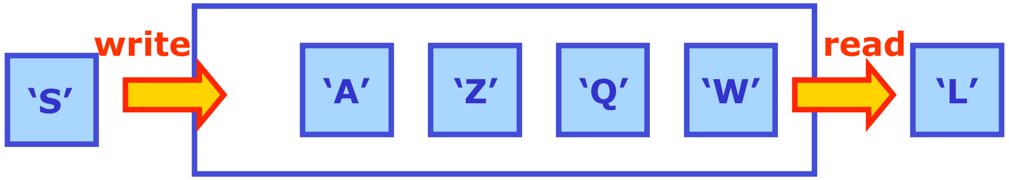
  - Support tasks
    1. How many characters are currently in the FIFO
    2. Is a FIFO empty
    3. Is a FIFO full

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

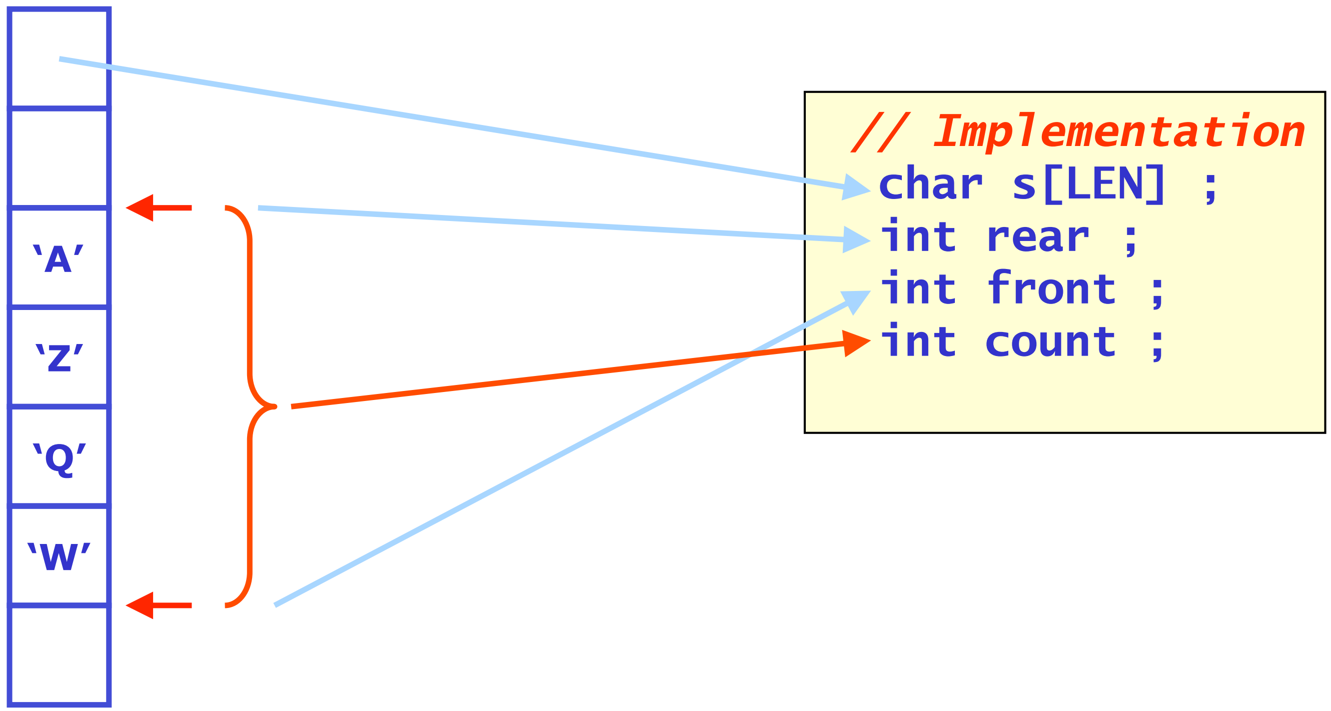
- List desired support tasks
  1. How many characters are currently in the FIFO
  2. Is a FIFO empty
  3. Is a FIFO full

```
// 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

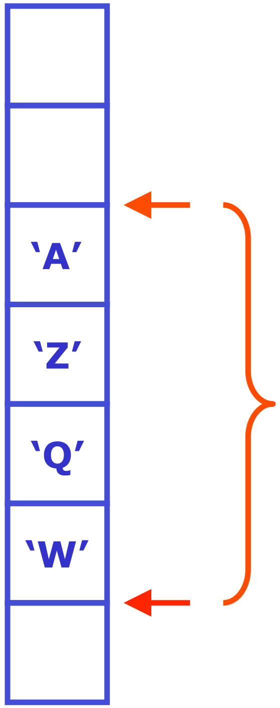


```
// Implementation
char s[LEN] ;
int rear ;
int front ;
int count ;
```

© 2006 Wouter Verkerke, NIKHEF

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

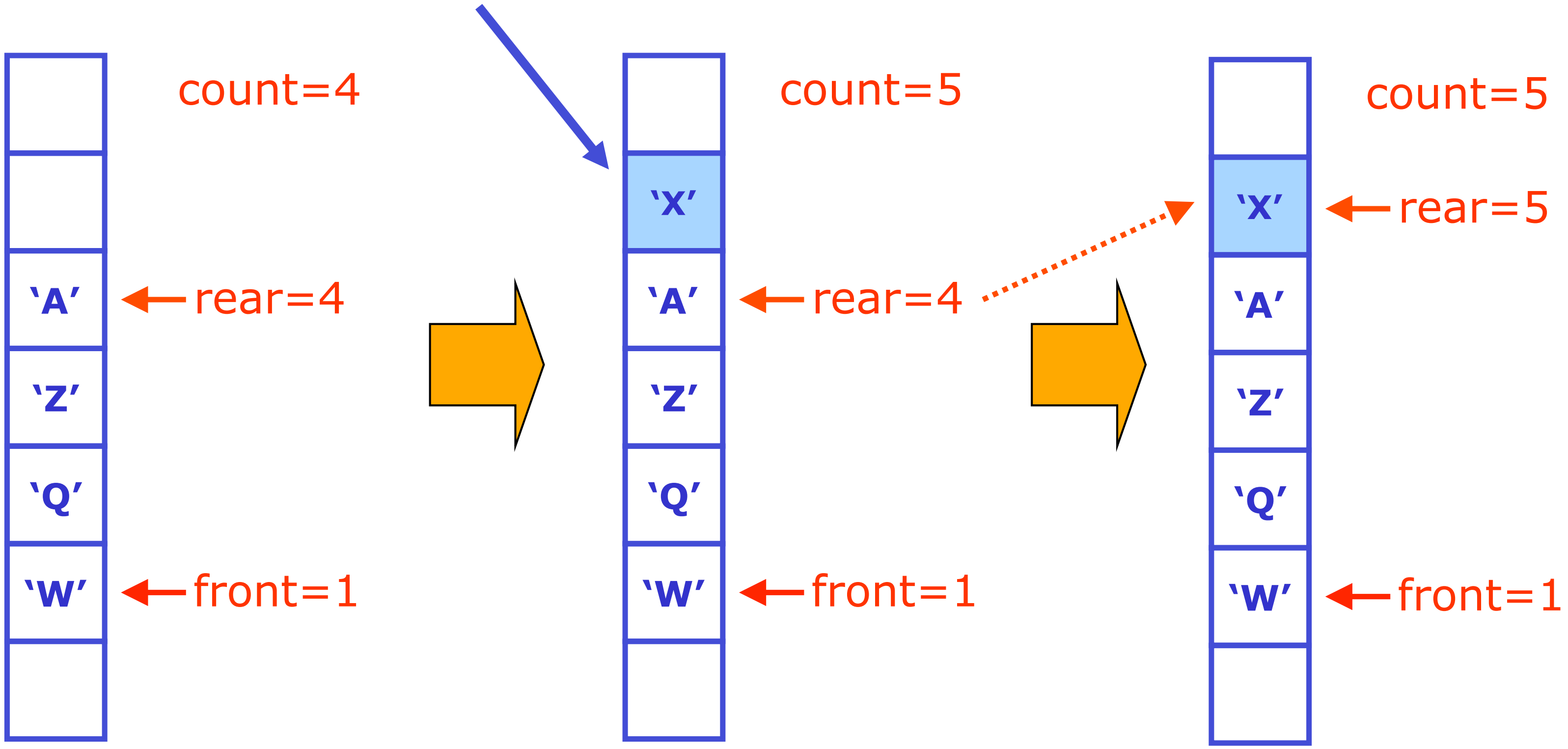


```
// Implementation
void init() { front = rear = count = 0 ; }

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

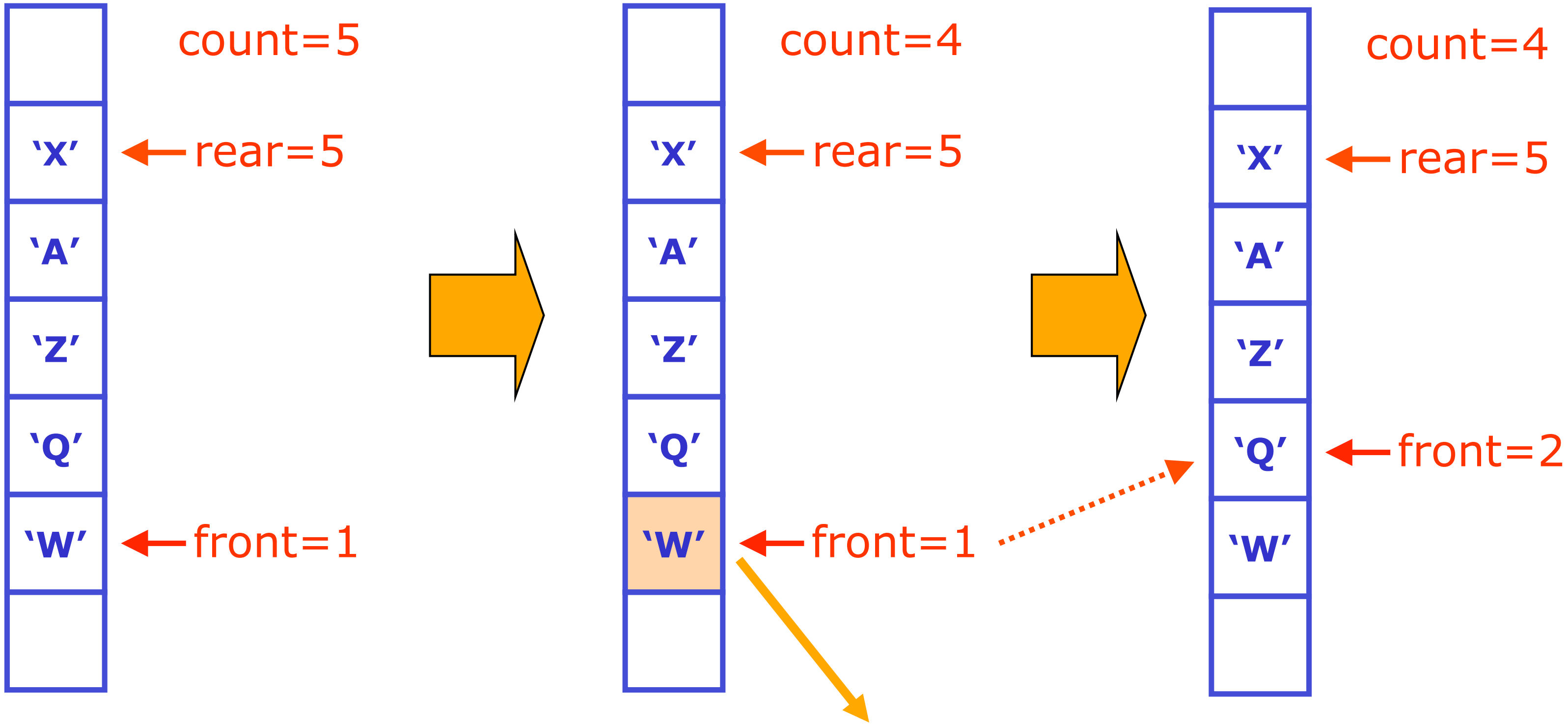
```
void write(char c) { count++ ;
                     if(rear==LEN) rear=0 ;
                     s[rear++] = c ; }
```



Designing the C++ struct FIFO – implementation

- Animation of FIFO read operation

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



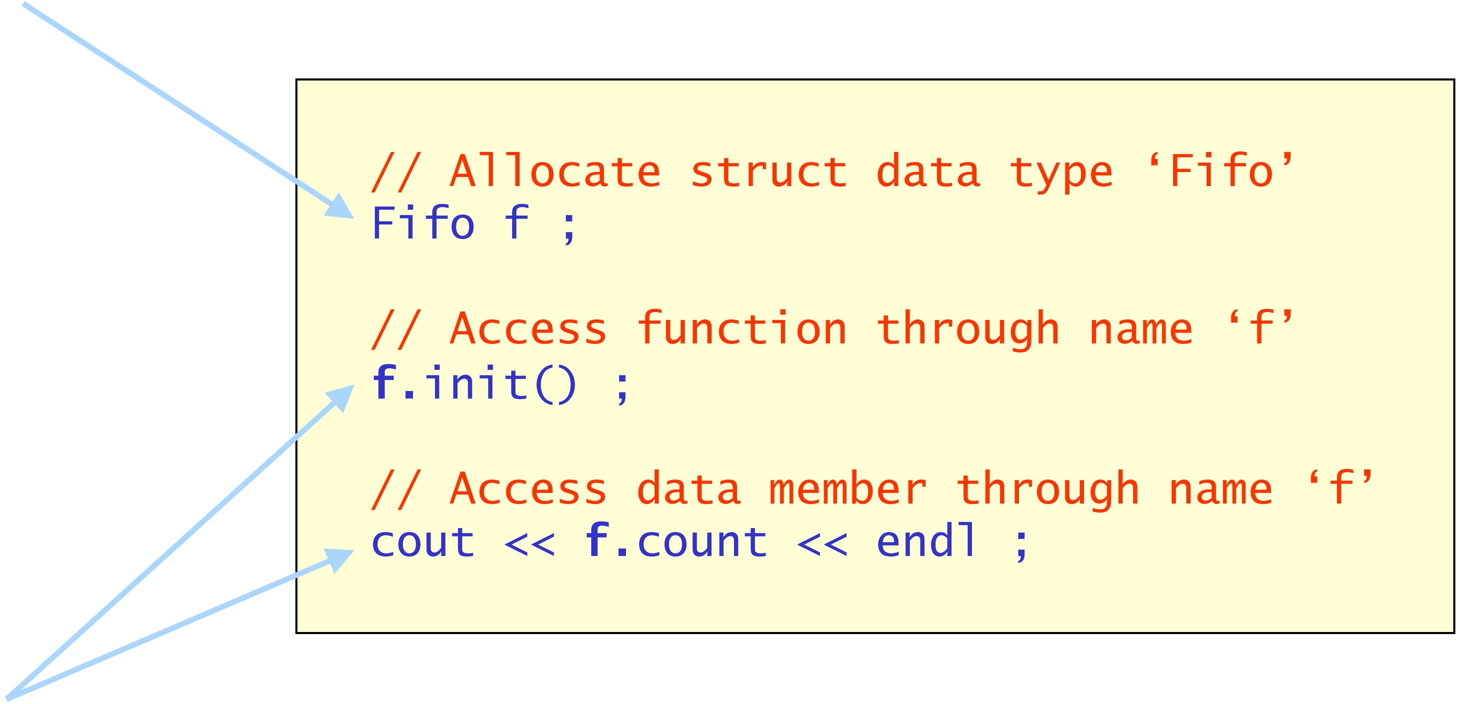
Putting the FIFO together – the struct concept

- 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

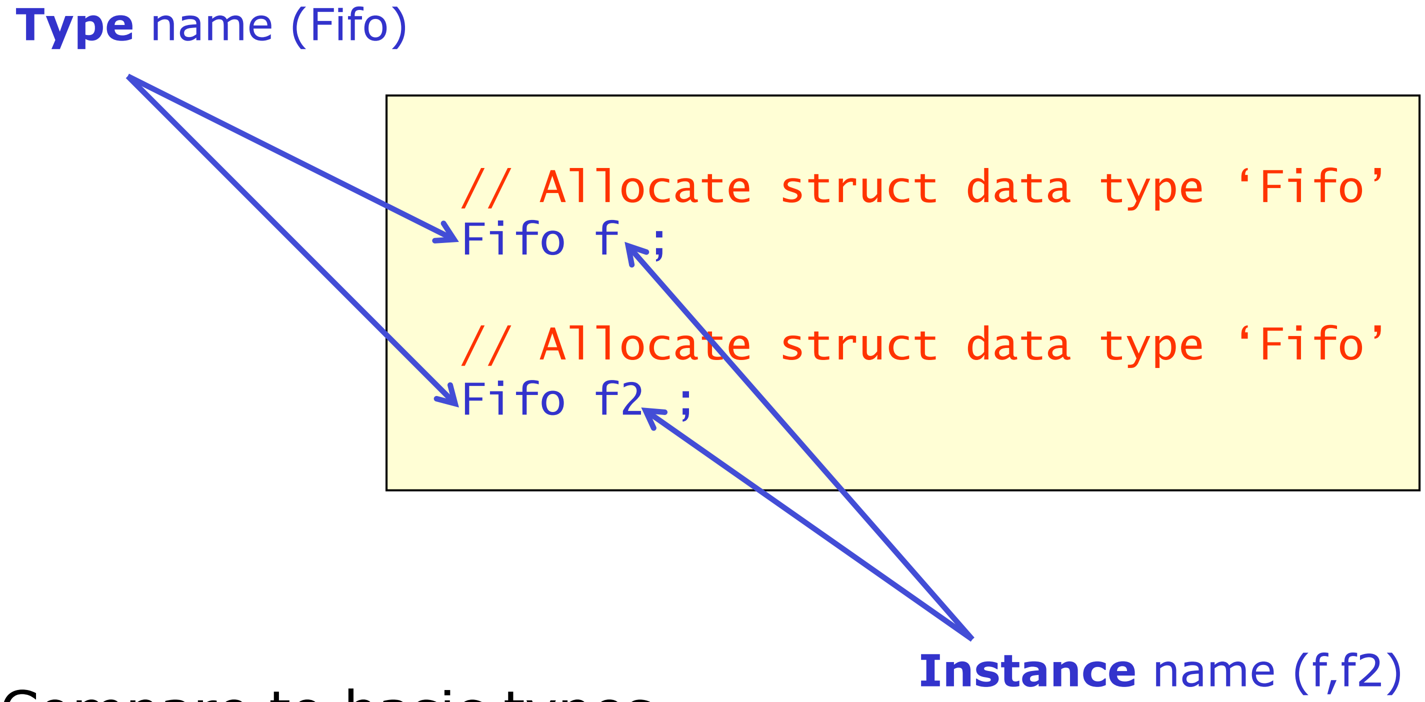


```
// Allocate struct data type ‘Fifo’
Fifo f ;

// 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

Type names vs. instance names

- Note important distinction between *type* name and *instance* name



Type name (Fifo)

```
// Allocate struct data type 'Fifo'
Fifo f ;

// Allocate struct data type 'Fifo'
Fifo f2 ;
```

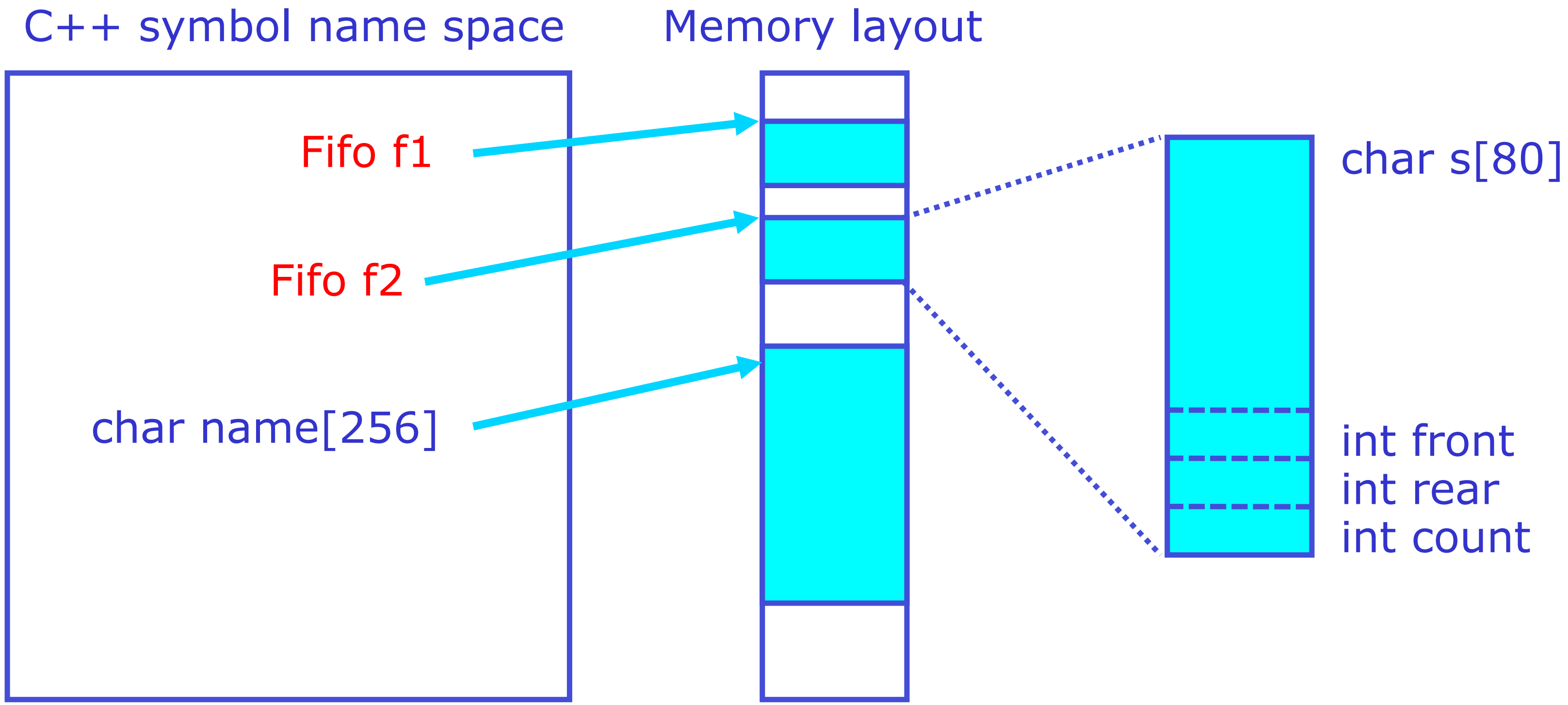
Instance name (f,f2)

- Compare to basic types

```
int i ;
int i2 ;
```

Type names vs. instance names

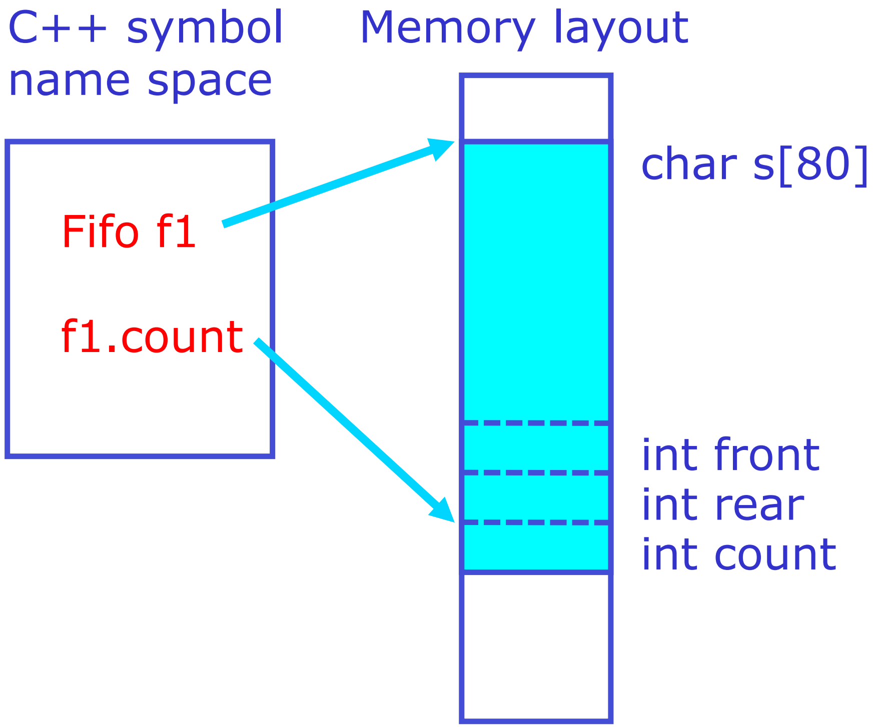
- *Instance* name (`f1,f2`) maps to address in memory
- *Type* name (`Fifo`) controls size of memory allocation, interpretation of memory in allocated block



Member access operator

- The dot (.) and arrow (->) operators implements access to members of composite object like struct's
  - Syntax: *TypeName.MemberName*

```
// Allocate struct
// data type 'Fifo'
Fifo f ;

// Access data member
// through name 'f'
cout << f.count << endl ;

// Access data member
// through pointer to f
Fifo* pf = &f ;
cout << (*pf).count << endl ;
cout << pf->count << endl ;
```



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 f1,f2 ;
read_fifo(f1) ;
read_fifo(f2) ;
```

```
// Solution with
// member functions

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

fifo f1,f2 ;
f1.read() ; // does f1.count--
f2.read() ; // does f2.count--
```

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 chars
in fifo
d
a
t
a

b
y
t
e
s
```

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 since 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 be accessed by member functions of the struct (which themselves can be either private or public)

Redesign of Fifo class with access restrictions

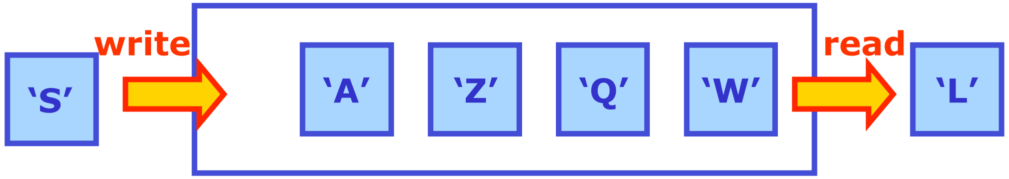
```
const int LEN = 80 ; // default fifo length

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 detail 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

```
struct Name {
private:

… members …

public:

… members …

} ;
```

Equivalent

⟺

```
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 declaration

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 type 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 object 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!

Data members vs function arguments

- Note that you can access two types of variables in class member functions, including the constructor
  - Data members – Will live beyond function call, but not beyond object lifetime
  - Function arguments – Will only for duration of function call

*If you need to preserve information given as function argument to constructor, you must **copy** it to a data member*

```
class Fifo {
public:

  Fifo(int size) { _size = size ;}

private:
  int _size ;
  …
```

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)

Class member initialization in C++2011

- In C++2011 a new intuitive form of data member initialization is supported: **assignment in the class declaration**

```
class Fifo {
  private:   // Implementation
  char s[LEN] ;
  int front = 0;
  int rear = 0 ;
  int count = 0;

  public:    // Interface
  …
} ;
```

  - Conceptually C++ compiler will translates assignments to corresponding member initializations ‘front(0) etc’

- If *both* assignment and ctor member initializer are specified, latter takes precedence
  - I.e. Assignment can be used as the ‘default’ initializer than can be overridden my member init in ctor

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 ;
 void 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

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

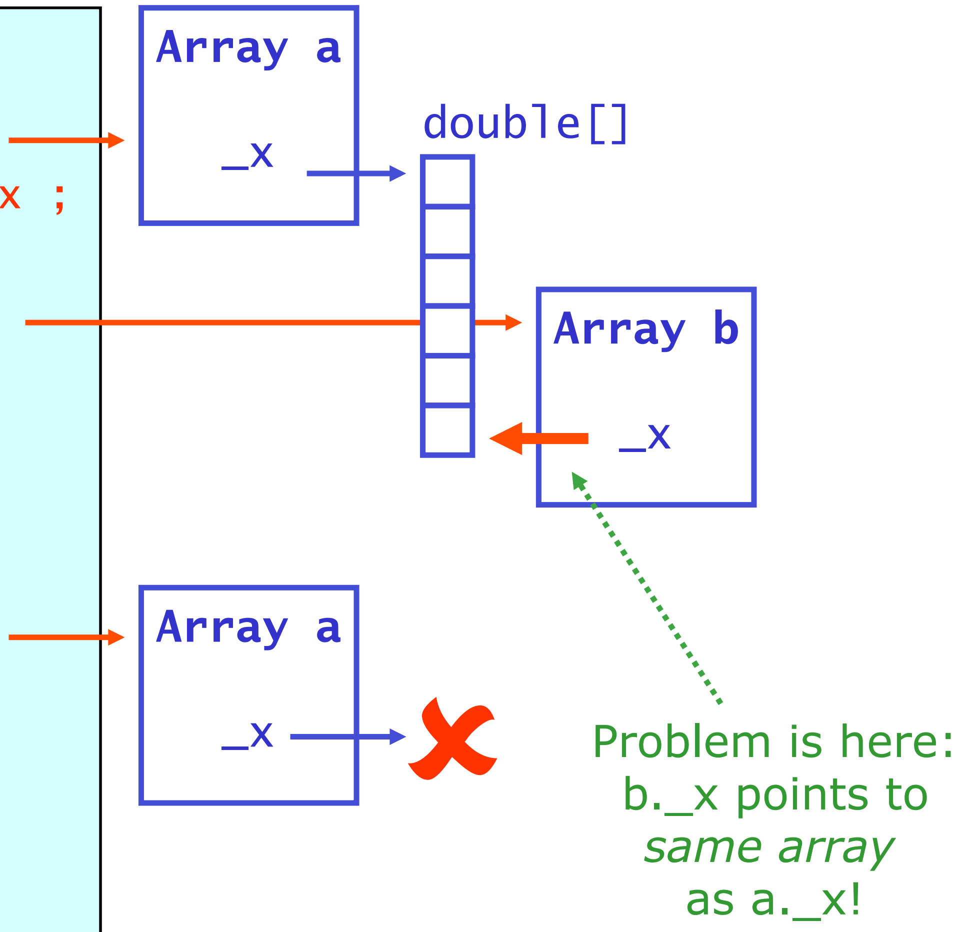
```
class Array {
public:
  Array(int size) {
    initialize(size) ;
  }
  ~Array() {
    delete[] _x ;
  }

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

- Example of default copy constructor wreaking havoc

```
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!
}
```



Problem is here:
b._x points to
same array
as a._x!

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 ;

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

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

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

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` instance tied to the same file makes 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() ; }
   …
} ;
```

Deleting default constructors in C++2011

- In C++2011 new language feature allows to delete default implementations of constructors explicitly as follows

```
class File {

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

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

   File(const File&) = delete ; // disallow copying

  ~File() { close() ; }
   …
} ;
```

4 Class Analysis & Design

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

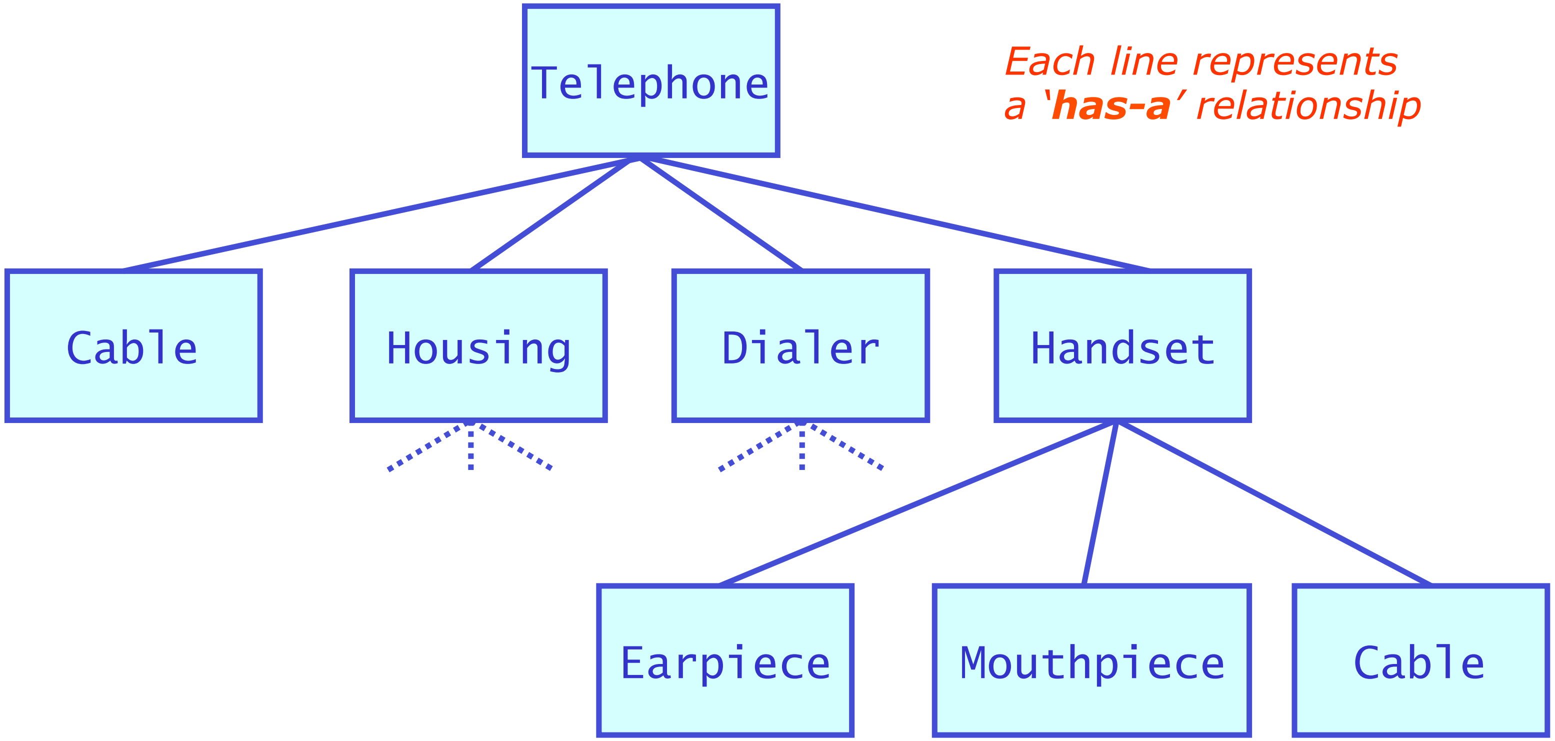
2. Associate software objects with natural objects (‘objects in the application domain’)
   - Actions translate to member functions
   - Attributes translate to data members

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

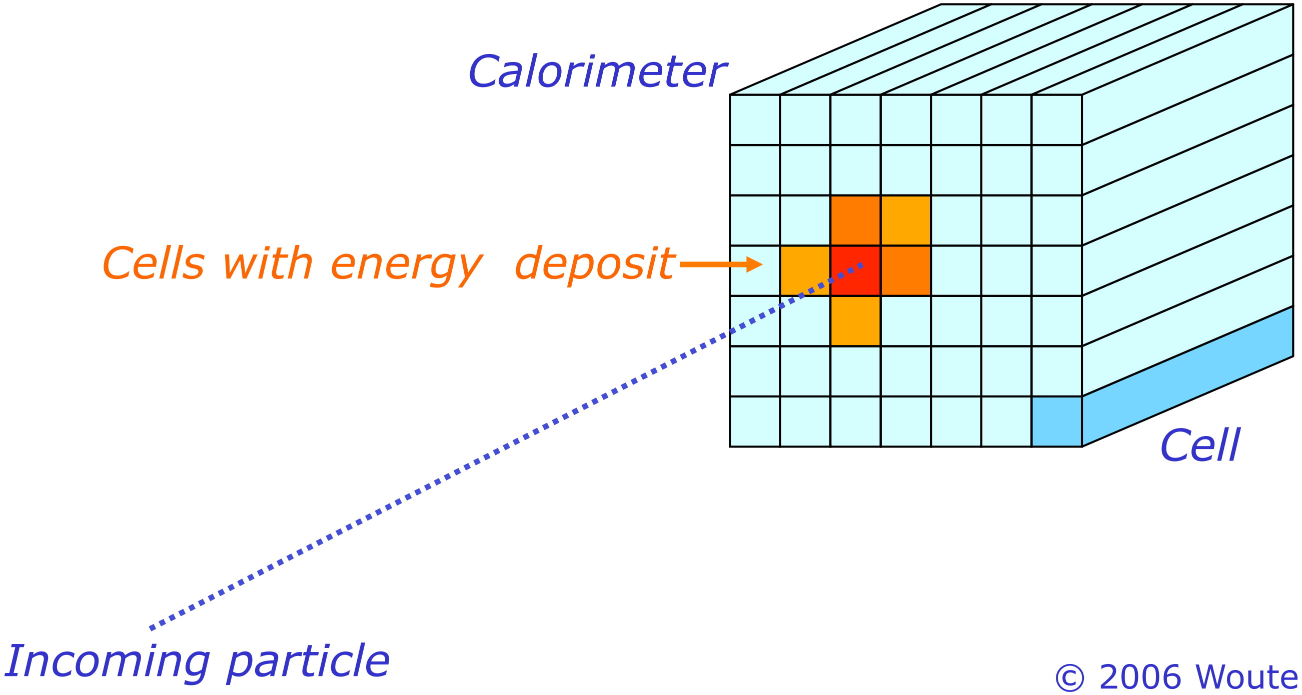


*Each line represents a ‘**has-a**’ relationship*

Analysis – Example from High Energy Physics

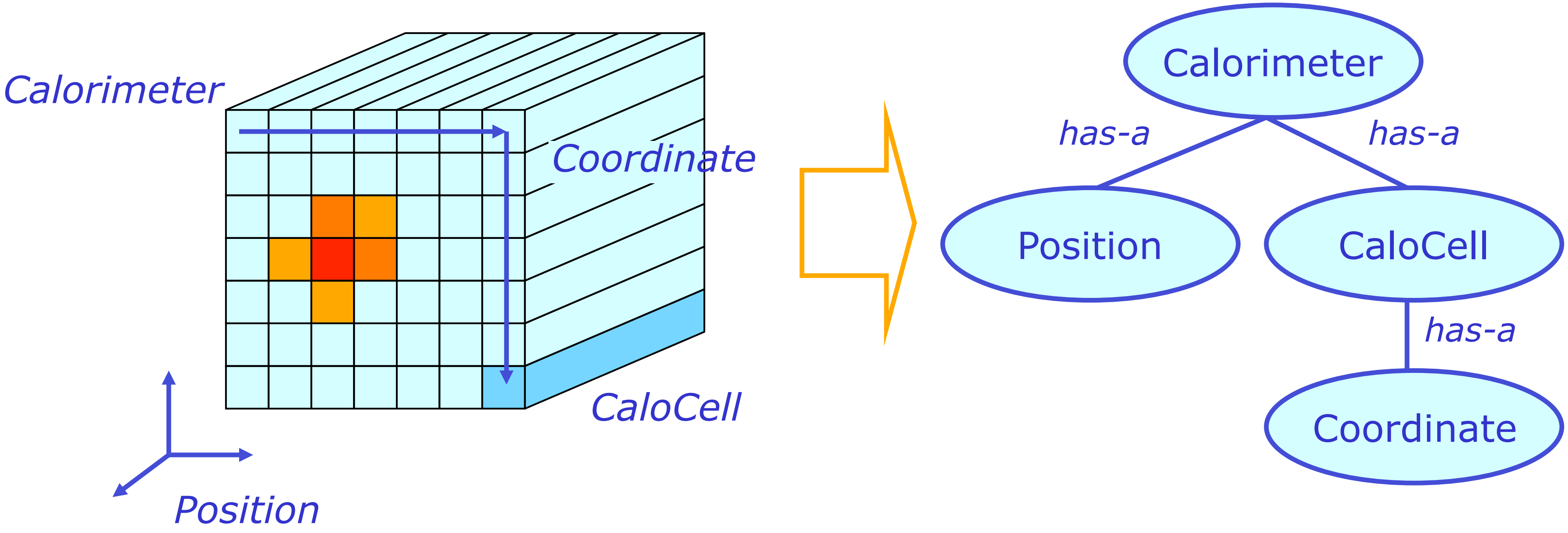
- Real life often not so clean cut

- 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



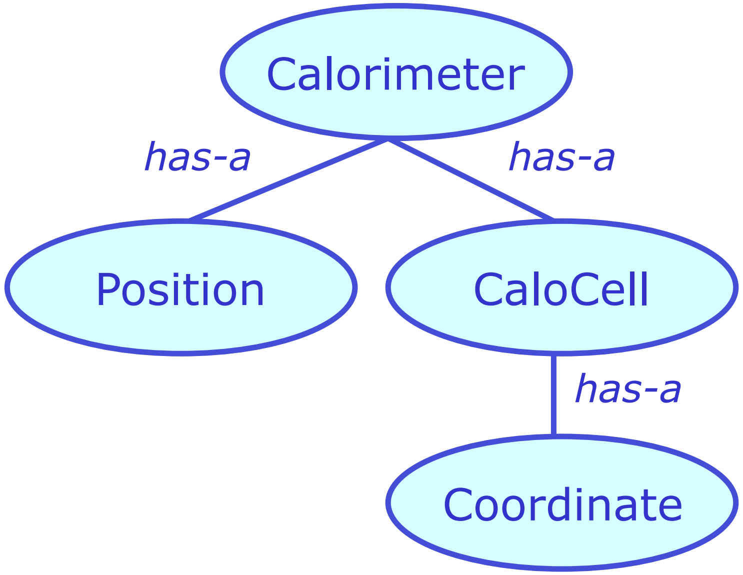
Analysis – Example from High Energy Physics

- 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



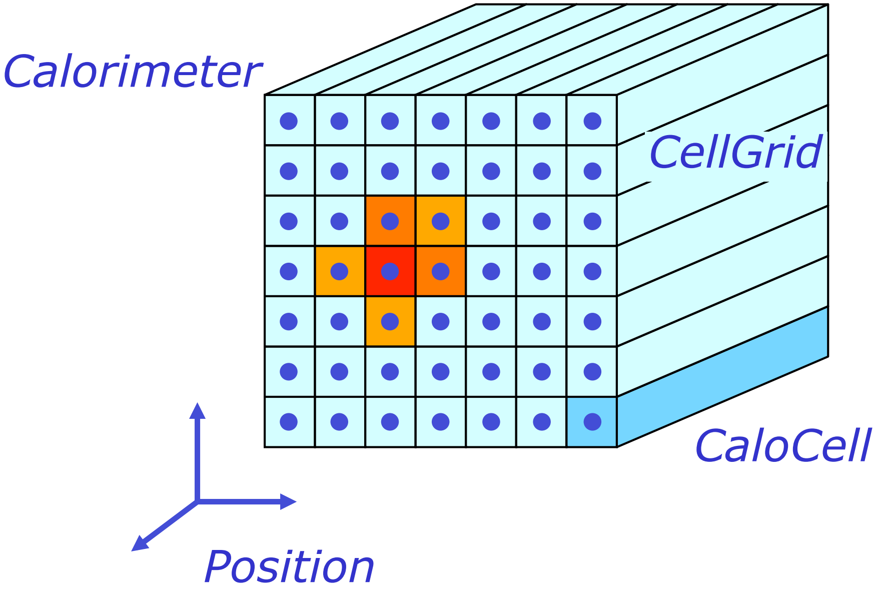
Analysis – Example from High Energy Physics

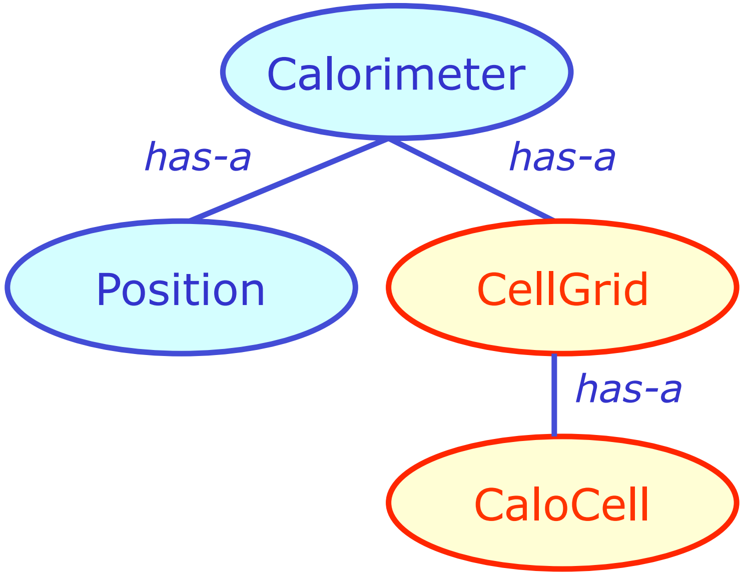
- Key Analysis sanity check – Can we describe what each object ***is***, in addition to what it does?
  - Answer: yes



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

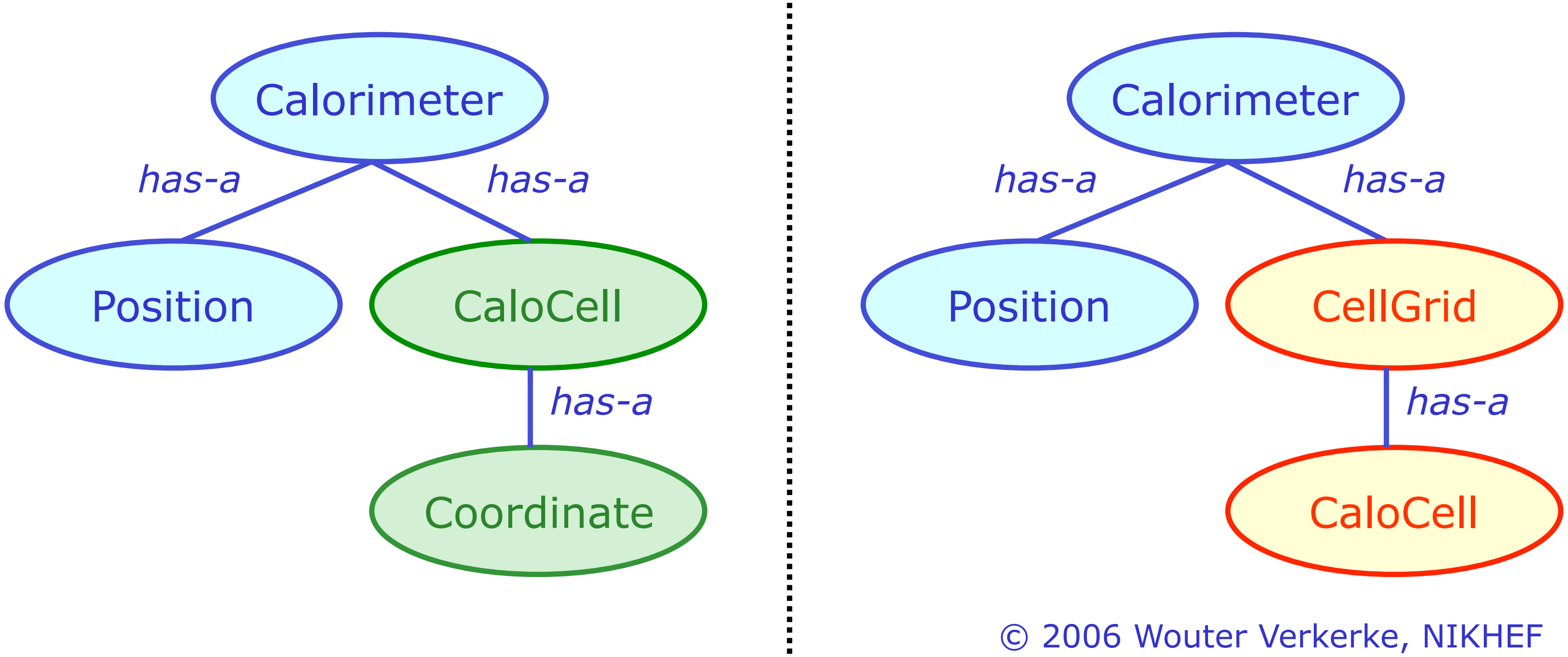




© 2006 Wouter Verkerke, NIKHEF

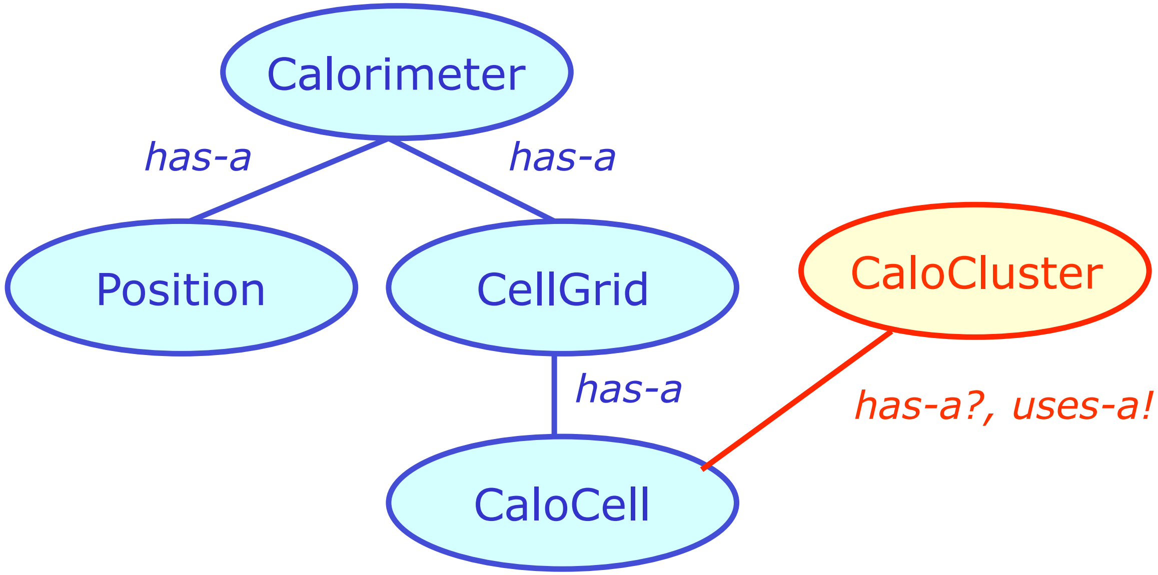
Analysis – Example from High Energy Physics

- 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**

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 ;      →      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 ;
   _len = other._len;
   _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 ;
```

→

$$\texttt{complex a,b,c ;}$$
$$\texttt{a.operator=(}\underbrace{\texttt{b.operator=(c))}}_{\text{Returns reference to b}}\texttt{ ;}$$

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() { if (_s) delete[] _s ; }

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

Data members, array & length

The custom string class

- Example string class for illustration of operator overloading

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

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

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

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

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

The custom string class

- Example string class for illustration of operator overloading

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

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

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

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

Ctor
Dtor

The custom string class

- Example string class for illustration of operator overloading

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

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

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

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

Overloaded assignment operator

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

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

    _s = newstr ;                       // install new buffer
    _len = newlen ;                     // set new length
    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

Working with class String

- Demonstration of operator+ use on class String

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

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

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

Implicit conversion by compiler

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

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

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()) ;
```

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
- Naïve – 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 automatically instantiates max(float&, float&)
cout << max(x,y) << endl ; // displays 5.6 (CORRECT)
```

Basics of 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 assumptions you make (if any)

Basics of 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 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 provides 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 ;                   Assumptions on TYPE
   int _len ;                   -Default constructor
   int _top ;                   -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 ;

   // OUTPUTS ‘2’

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

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

   // OUTPUTS ‘World Hello’

}
```

Initializer list of generic containers (C++ 2011)

- In C++2011 the compound initializer syntax of arrays can be extended to generic container classes

```
int x[3] = { 0, 1, 2 } ;

IntVector iv = { 0, 1, 2 } ; // Also works!

// Because constructor with initializer_list
// was added to class IntVector

class IntVector {
public:
  IntVector(std::initializer_list<int> ilist) ;
  ~IntVector() ;

private:
  int* _xvec ;
} ;
```

Initializer list of generic containers (C++ 2011)

- In C++2011 the compound initializer syntax of arrays can be extended to generic container classes
  - Retrieve content with iterator semantics – more in Module 7

```
class IntVector {
public:
  IntVector(std::initializer_list<int> ilist) {

    _xvec = new int[ilist.size()] ;

    int i(0) ;
    auto iter = ilist.begin() ;
    while (iter != ilist.end()) {
        _xvec[i++] = *iter ;
        iter++ ;
  }
  ~IntVector() ;

private:
  int* _xvec ;
} ;
```

Pointer memory management tools (C++2011)

- C++ also adds templated-based tools for pointer-based memory management
- Idea: have a dedicated wrapper class that ‘owns’ a pointer
  - Can be returned by-value from functions, if wrapper is deleted because it goes out of scope, it will delete the pointer
- Situation without wrapper

```
double* allocate_buffer(int size) {
    return new double[size] ;
}

int main() {
   // we own tmp, don’t forget to delete
   double* tmp = allocate_buffer(100) ;
   tmp[3] = 5 ;
}
```

Pointer memory management tools (C++2011)

- C++ also adds templated-based tools for pointer-based memory management
- Idea: have a dedicated wrapper class that ‘owns’ a pointer
  - Can be returned by-value from functions, if wrapper is deleted because it goes out of scope, it will delete the pointer
- Situation with wrapper

```
unique_ptr<double> allocate_buffer(int size) {
    return unique_ptr<double>(new double[size]) ;
}

int main() {
   // we own tmp, don’t forget to delete
   unique_ptr<double> tmp = allocate_buffer(100) ;
   tmp[3] = 5 ;
}
// memory held by tmp deleted when tmp goes out of scope
```

Pointer memory management tools (C++2011)

- C++ also adds templated-based tools for pointer-based memory management
- Idea: have a dedicated wrapper class that ‘owns’ a pointer
  - Can be returned by-value from functions, if wrapper is deleted because it goes out of scope, it will delete the pointer
- Situation with wrapper

```
int main() {
   // we own tmp, don’t forget to delete
   unique_ptr<double> tmp = allocate_buffer(100) ;
   tmp[3] = 5 ;
}
```

Class unique_ptr overloads operator-> to return pointer to payload. Can use unique_ptr<T> in same way as T*

7 Standard Library II the Template Library

Introduction to STL

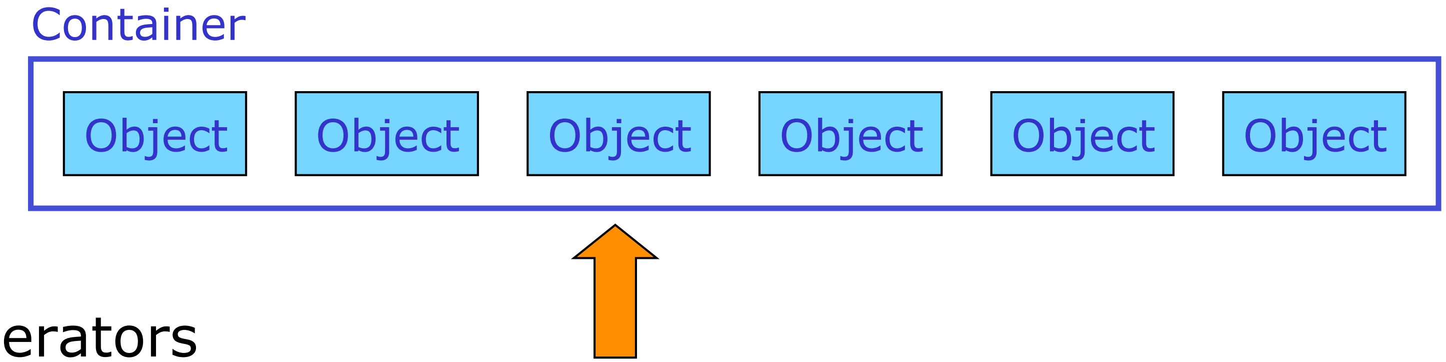
- STL = The **S**tandard **T**emplate **L**ibrary
  - A collection of template classes and functions for general use
  - Started out as experimental project by Hewlett-Packard
  - Now integral part of ANSI C++ definition of 'Standard Library'
  - Excellent design!

- 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

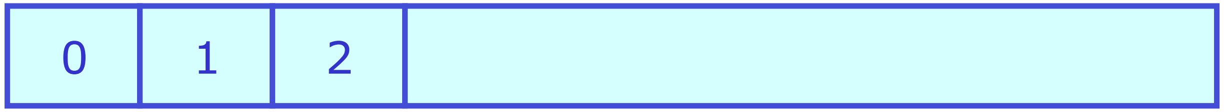
- Associative containers (no defined order, access by key)
  - `set`
  - `multiset`
  - `map`
  - `Multimap`
  - `unordered_set, unordered_map (C++2011)`

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



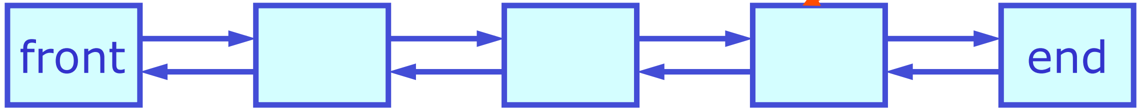
  - Manages its own memory allocation
  - Initial length at construction, but can be extended later
  - Elements initialized with default constructor
  - **Offers fast random access to elements**
  - Example

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

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

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

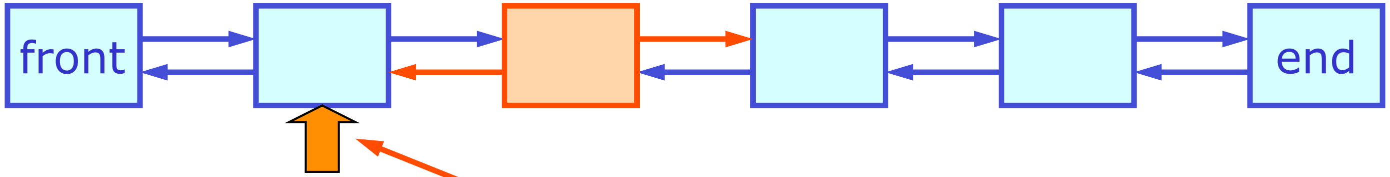
List <list>

```
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

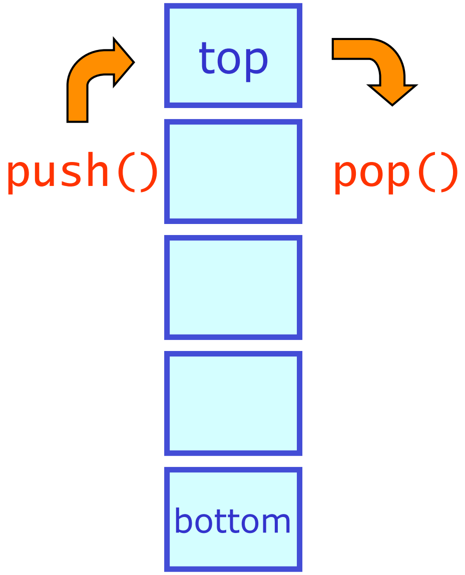
iterator ‘pointer’ in collection

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

Stack <stack>

- A stack is an adapter of deque
  - It provides a restricted view of a deque
  - Can only insert/remove 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<string>& s) {
   while(!s.empty()) cout << s.pop() << " " ;
}

// outputs "Mies Noot Aap"
```



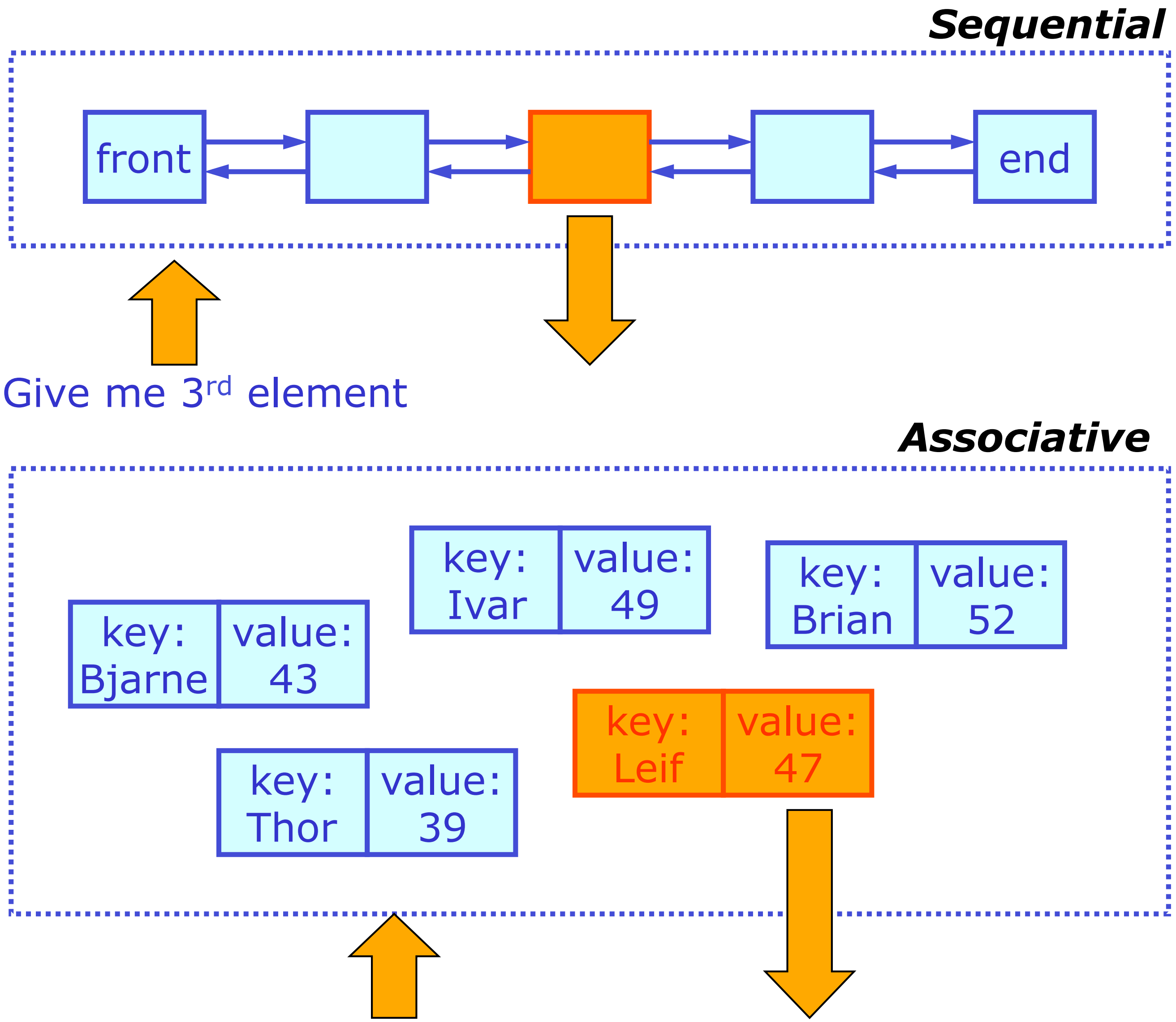
Sequential versus associative containers

- 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] = 4rd element of vector v

List<double> 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



Sequential

front

end

Give me 3rd element

Associative

key: Ivar | value: 49

key: Brian | value: 52

key: Bjarne | value: 43

key: Leif | value: 47

key: Thor | value: 39

Give me **value** of element
with **key** “Leif”

© 2006 Wouter Verkerke, NIKHEF

Pair <utility>

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

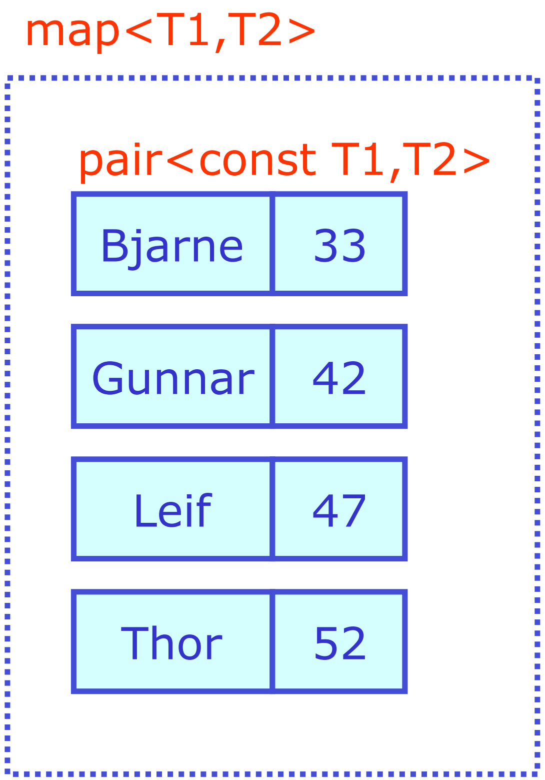
```
template<type T1, type T2>
struct pair {
   T1 first ;
   T2 second ;
   pair(const T1&, const T2&) ;
} ;

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<T1,T2>

pair<const T1,T2>

Bjarne 33

Gunnar 42

Leif 47

Thor 52

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”] ; // another = 0
```

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

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 different from a regular C++ pointer? – An iterator is aware of the collection it is bound to*.
  - *How do 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 array[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 array[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 array[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 array[10] ;                  vector<double> v(10) ;

int i = 0 ;
double* ptr ;                       vector<double>::iterator iter ;

ptr = &array[0] ;                   iter = v.begin() ;

while(i<10) {                       while(iter!=v.end()) {

  cout << *ptr << endl ;              cout << *iter << endl ;

  ++ptr ;                             ++iter ;
  ++i ;                             }
}
```

Check if you're at the end of your array

Check if you're at the end of your vector

Taking a more abstract view of containers

- Illustration of iterators vs C++ pointers

```
double array[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 array[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

Auto types work great with STL contains C++2011

- Note that ‘auto’ types are particularly handy when using STL classes as iterator type names are usually long, and never explicitly needed

```
// Iterator loop
vector<int> v(10) ;
vector<int>::iterator iter ;
for (iter=v.begin() ;iter!=v.end() ; ++iter) {
   *iter = 0 ;
}
```

```
// Iterator loop
vector<int> v(10) ;
for (auto iter=v.begin() ; iter!=v.end() ; ++iter) {
   *iter = 0 ;
}
```

Even better: range-based for loops C++2011

- C++2011 also introduces concept of ‘range-based’ for loops over any entity that supports iterators

```
std::vector<int> v = {0, 1, 2, 3, 4, 5};

// Loop over all elements of v
for (auto i : v) { // access by value,
    cout << i << endl ;
}

// Loop over all elements of v
for (auto&& i : v) { // access by reference,
 cout << i << endl ;
}
```

- Works for any container that defines methods begin() and end() that return an iteratable type

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

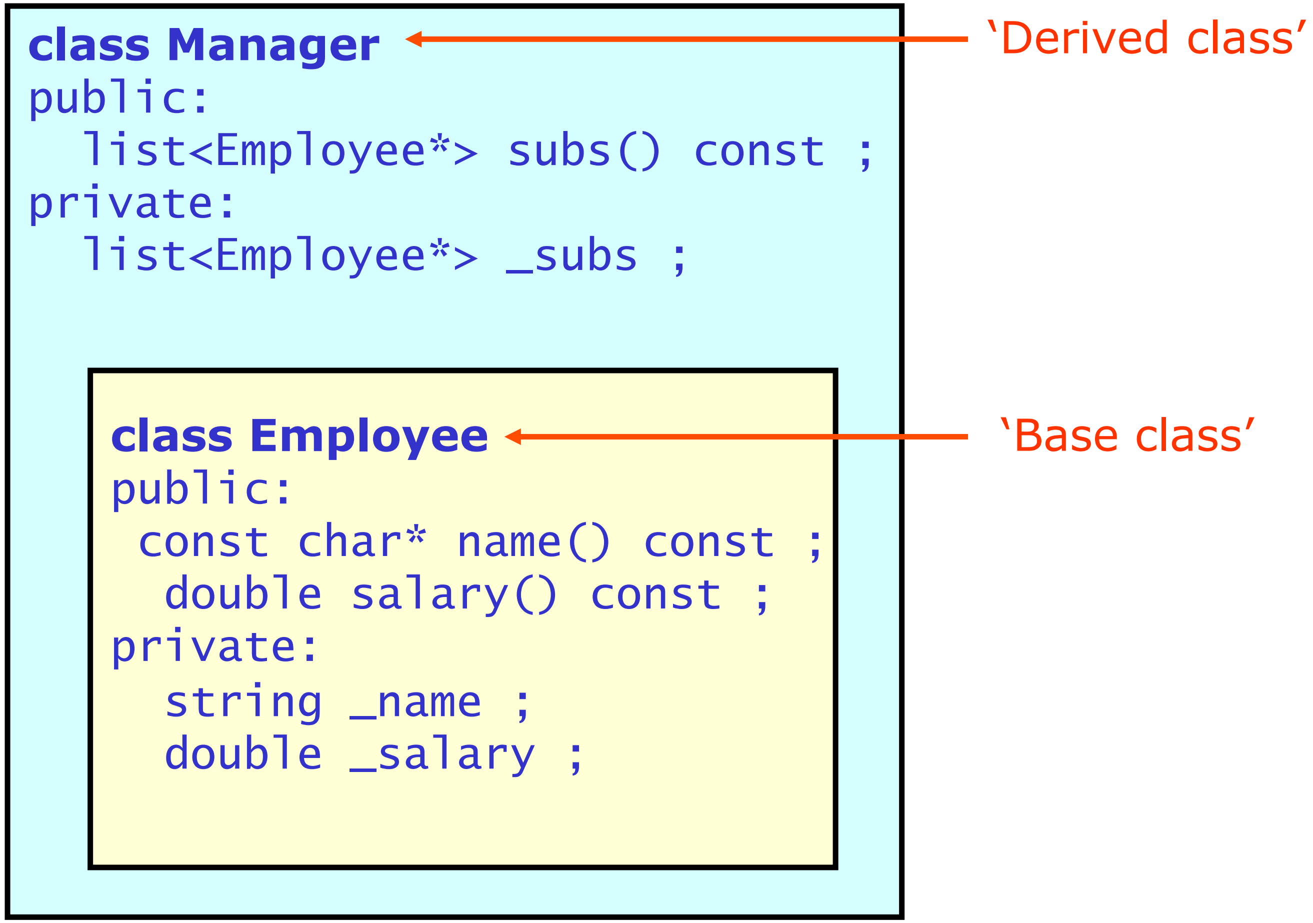
- Inheritance means: Manager **Is-A**n Employee
  - Object of class Manager can be used in exactly the same way as you would use an object of class Employee because:
  - class Manager also has all data members and member functions of class Employee
  - Detail: examples shows '*public inheritance*' – 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



```
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’

‘Base class’

Inheritance – Using it

- Demonstration of Manager-IS-Employee concept

```
// Create employee, manager record
Employee* emp = new Employee(“Wouter”,10000) ;

list<Employee*> subs ;
subs.push_back(emp) ;

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 also a pointer to the base class

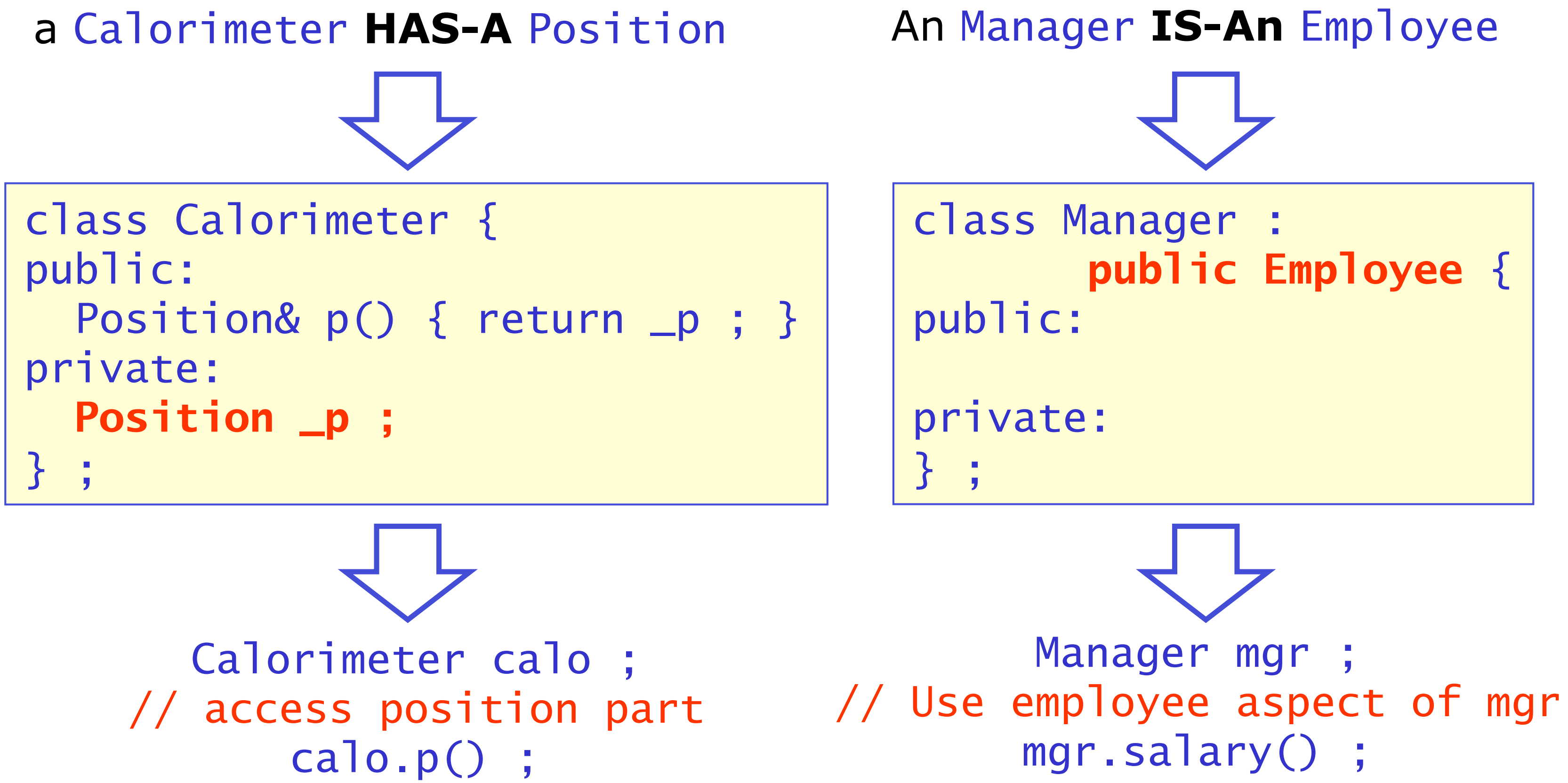
```
// Pointer-to-derived IS Pointer-to-base
void processEmployee(Employee& emp) {
   cout << emp.name() << “ : “ << emp.salary() << endl ;
}

processEmployee(*emp) ;
processEmployee(*mgr) ; // OK Manager IS Employee
```

  - But the 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)
  - A 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 – 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

Polymorphism

- Polymorphism is the ability of an object to retain its true identity even when accessed 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 accessed 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 to 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

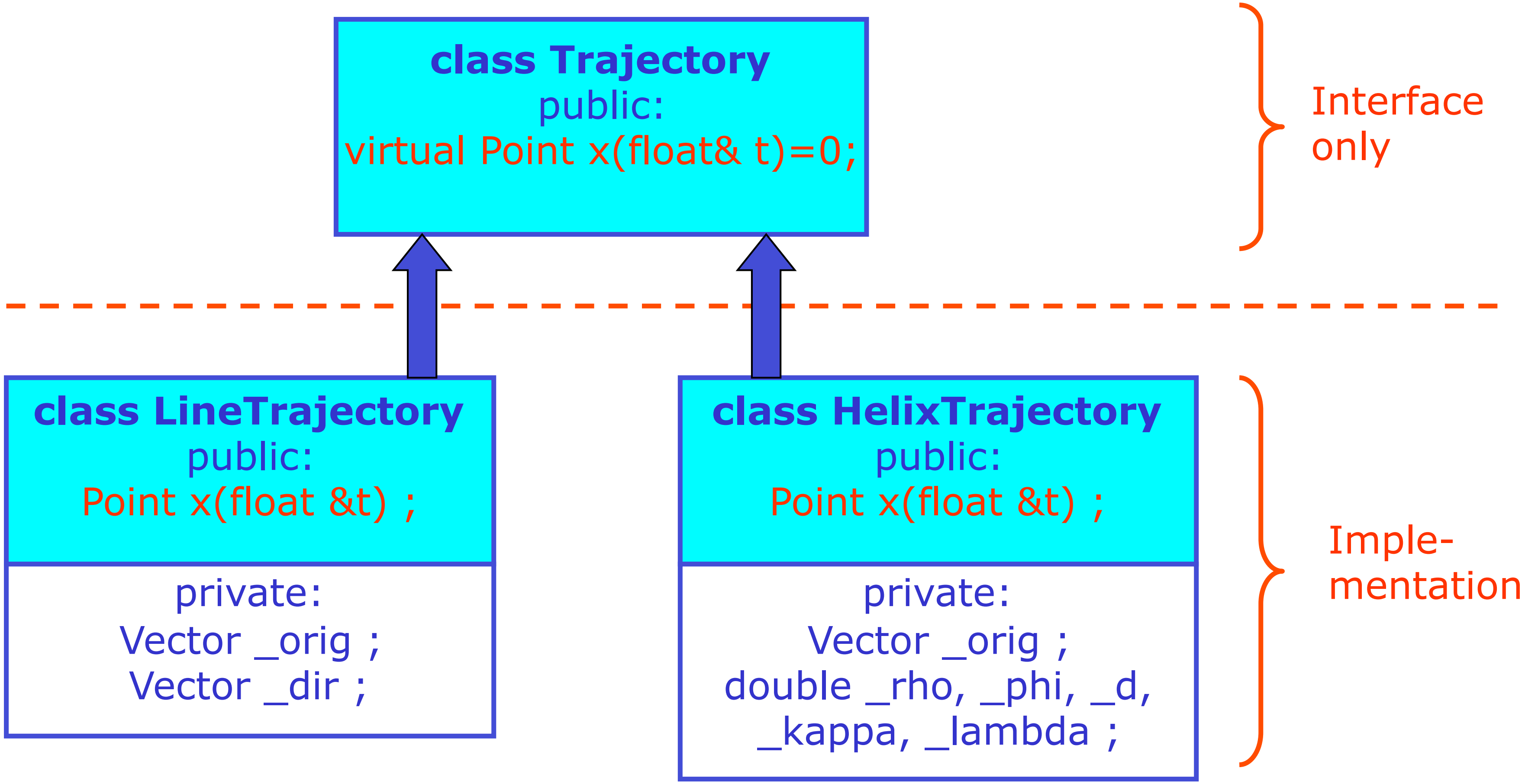
- Watch the destructor declaration if you define virtual functions
  - Example

```
Employee* emp = new Employee(“Wouter”,10000) ;
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



class Trajectory
public:
virtual Point x(float& t)=0;

Interface only

class LineTrajectory
public:
Point x(float &t) ;
private:
Vector _orig ;
Vector _dir ;

class HelixTrajectory
public:
Point x(float &t) ;
private:
Vector _orig ;
double _rho, _phi, _d,
_kappa, _lambda ;

Imple-
mentation

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 implementations of a common 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

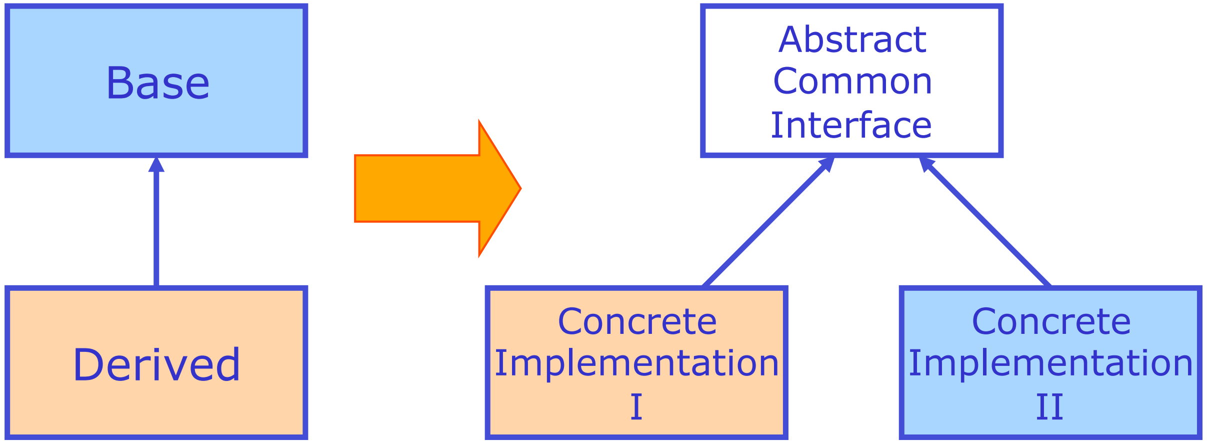
- Example on how to use abstract base classes

```
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)

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


Base
Derived
Abstract Common Interface
Concrete Implementation I
Concrete Implementation II