Object-based programming – Classes

3 Class Basics

Overview of this section

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

Encapsulation

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

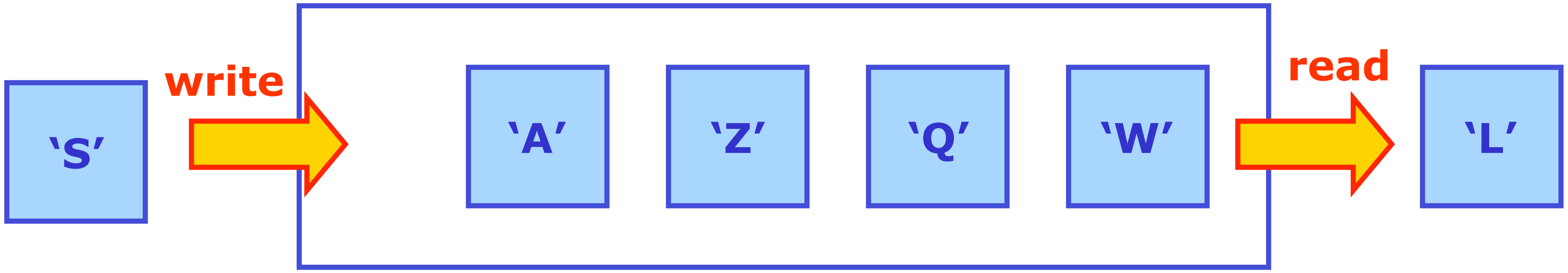
- Example

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

Evolving code design through use of C++ classes

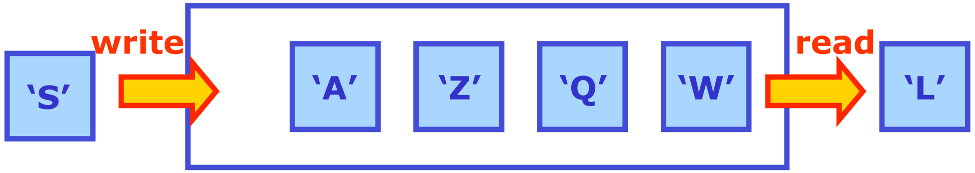
- Illustration of utility of C++ classes – Designing and building a FIFO queue
  - FIFO = '**F**irst **I**n **F**irst **O**ut'

- Graphical illustration of a FIFO queue



Evolving code design through use of C++ classes

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



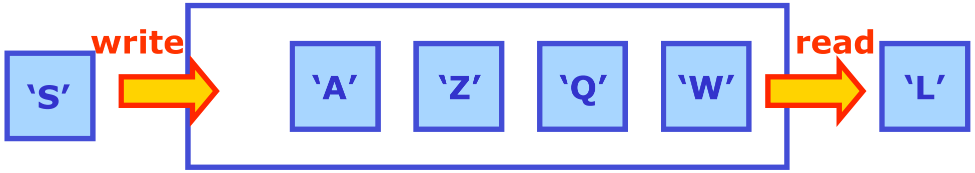
- List the essential interface tasks
  1. **Create** and initialize a FIFO
  2. **Write** a character in a FIFO
  3. **Read** a character from a FIFO
  - Support tasks
    1. How many characters are currently in the FIFO
    2. Is a FIFO empty
    3. Is a FIFO full

Designing the C++ class FIFO – interface

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

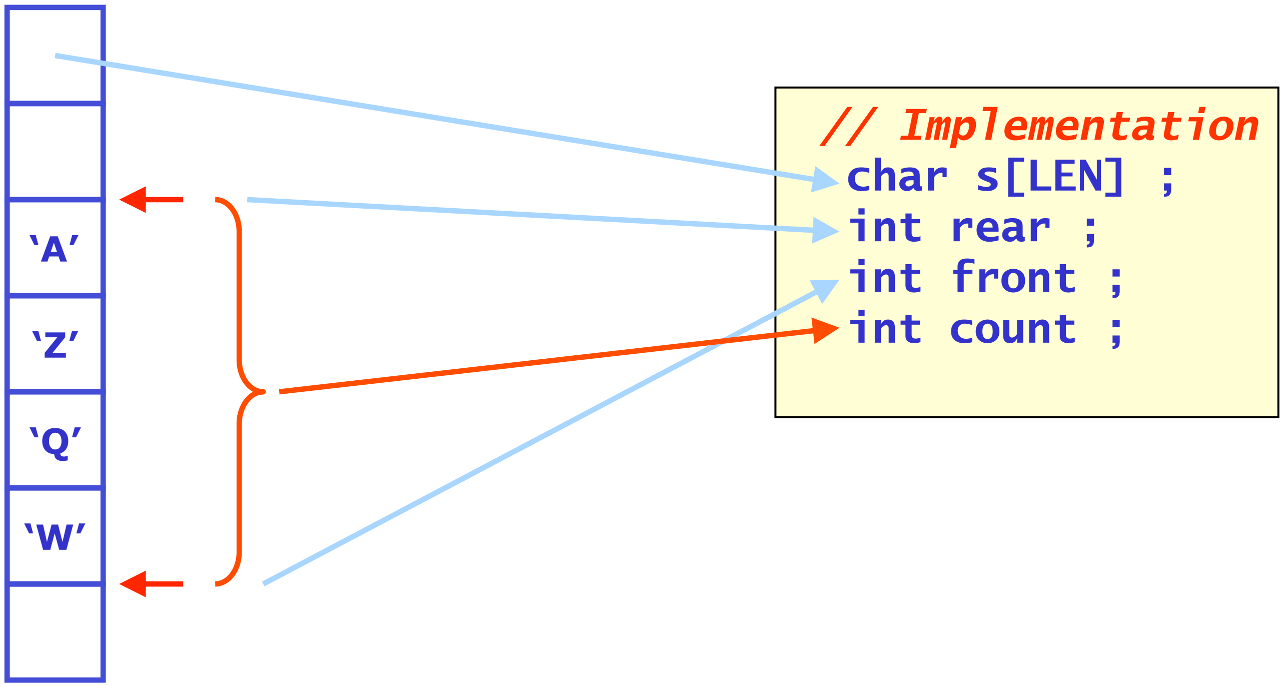
- List desired support tasks
  1. How many characters are currently in the FIFO
  2. Is a FIFO empty
  3. Is a FIFO full

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

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



Designing the C++ struct FIFO – implementation

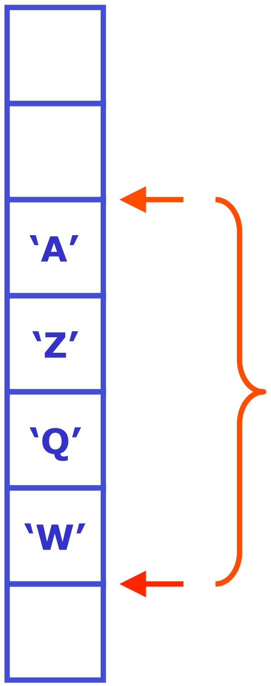
- Implement FIFO with array of elements
  - Use index integers to keep track of front and rear, size of queue



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

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

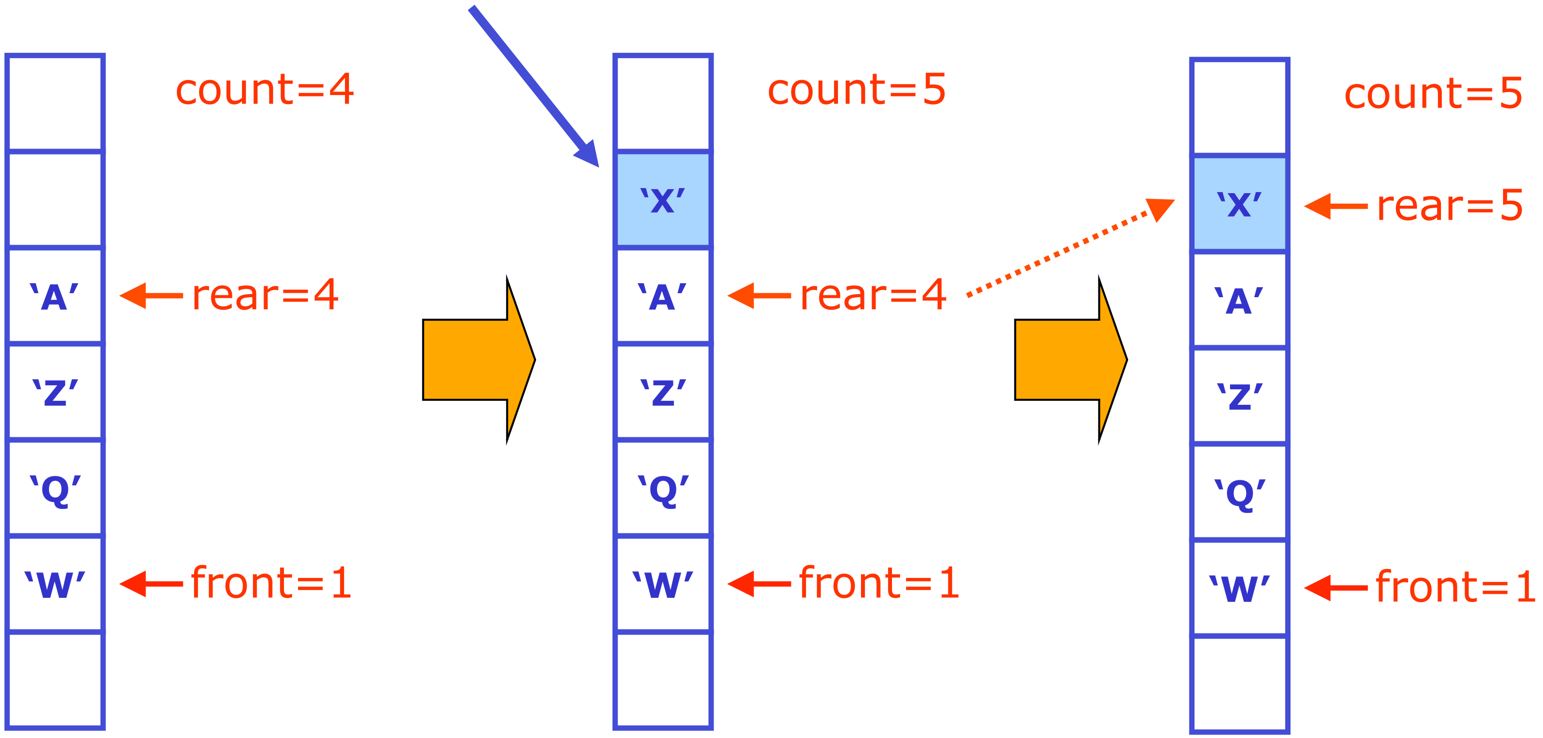


```
// Implementation
void init() { front = rear = count = 0 ; }

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

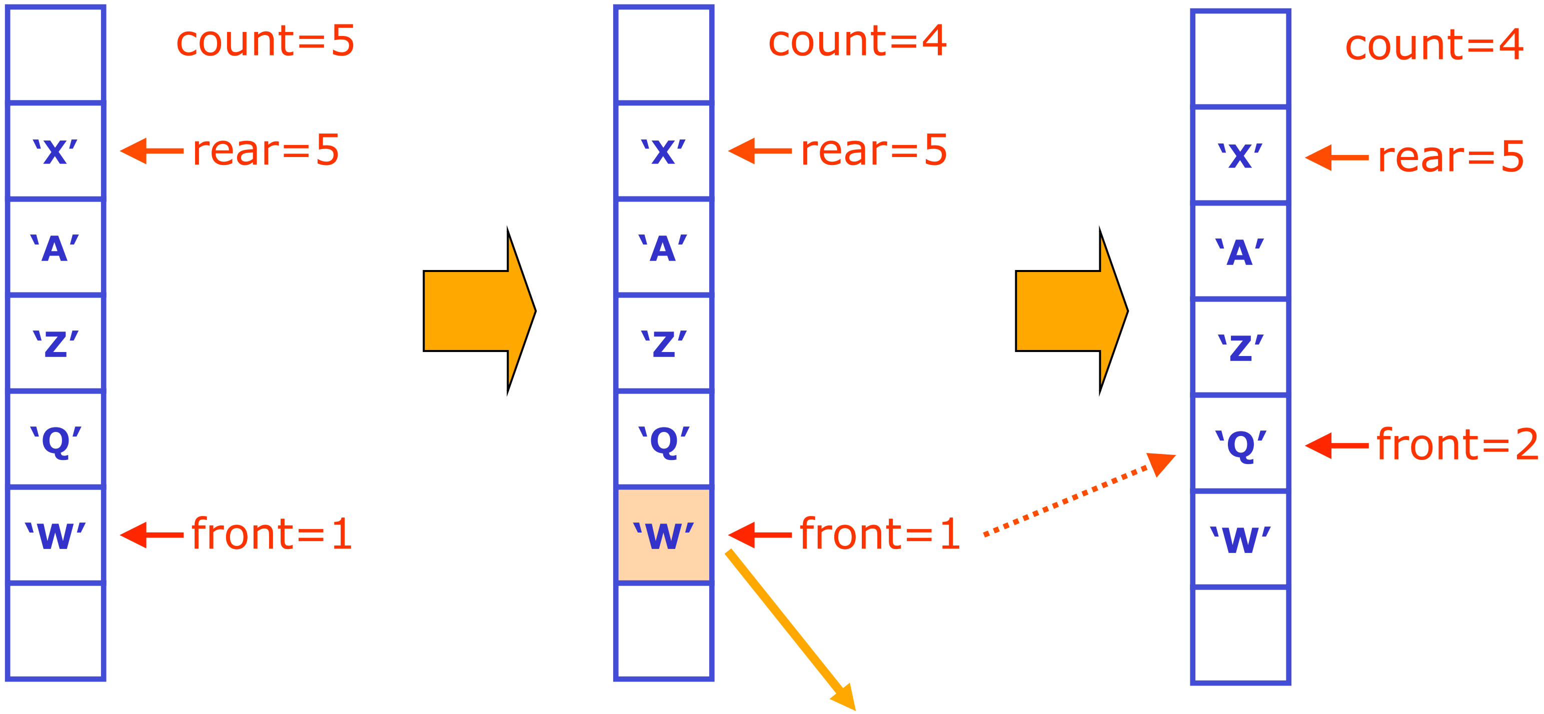
```
void write(char c) { count++ ;
                     if(rear==LEN) rear=0 ;
                     s[rear++] = c ; }
```



Designing the C++ struct FIFO – implementation

- Animation of FIFO read operation

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



© 2006 Wouter Verkerke, NIKHEF

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

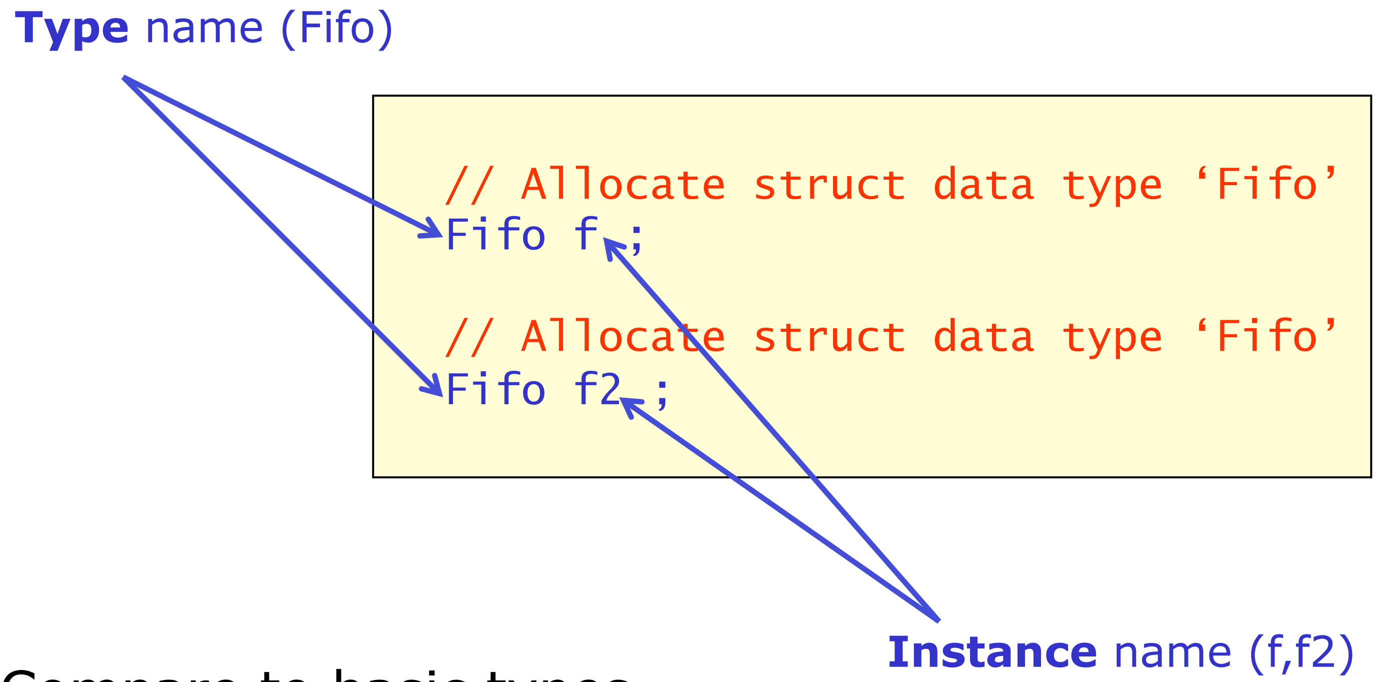
- Grouping of data members facilitates storage allocation
  - Single statement allocates all data members

```
// 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 ;
```

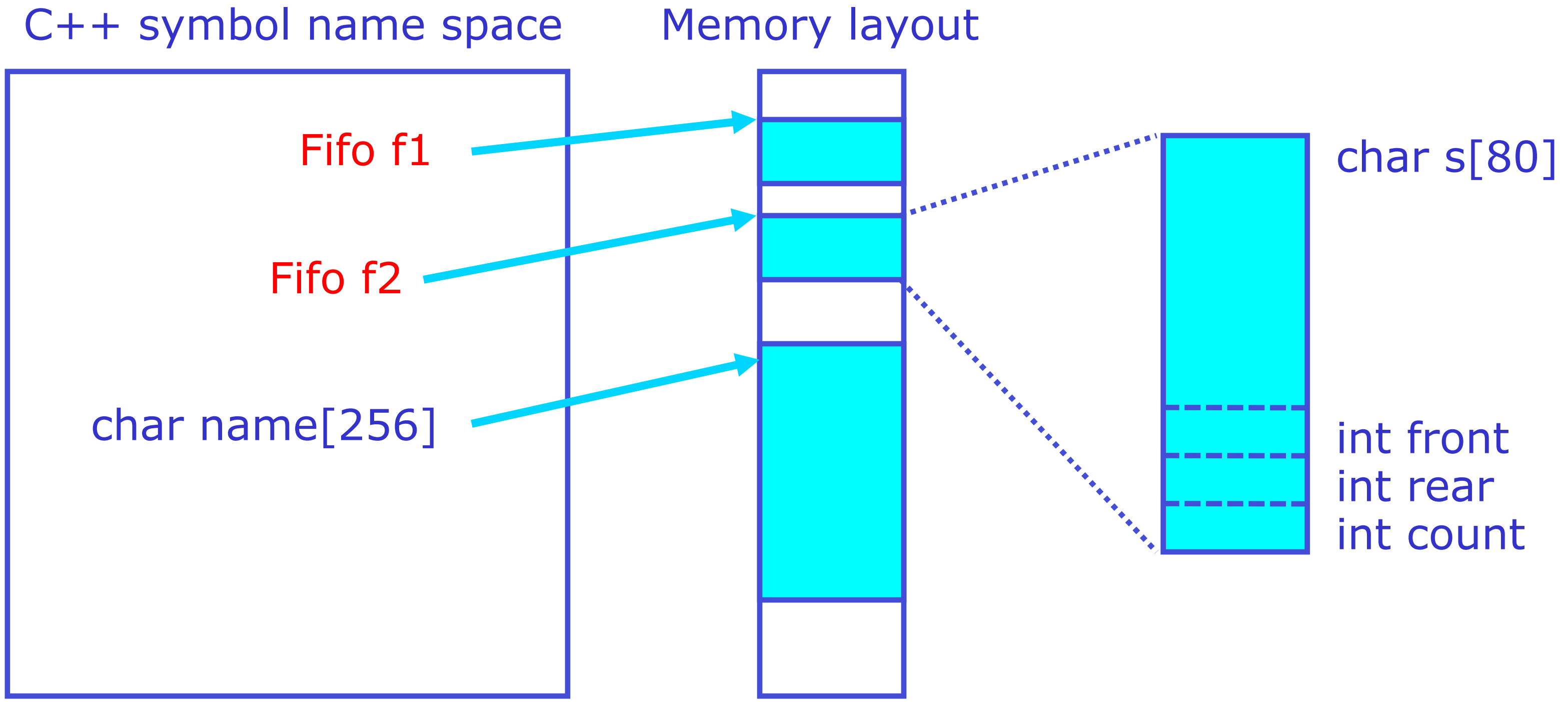
Instance name (f,f2)

- Compare to basic types

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

Type names vs. instance names

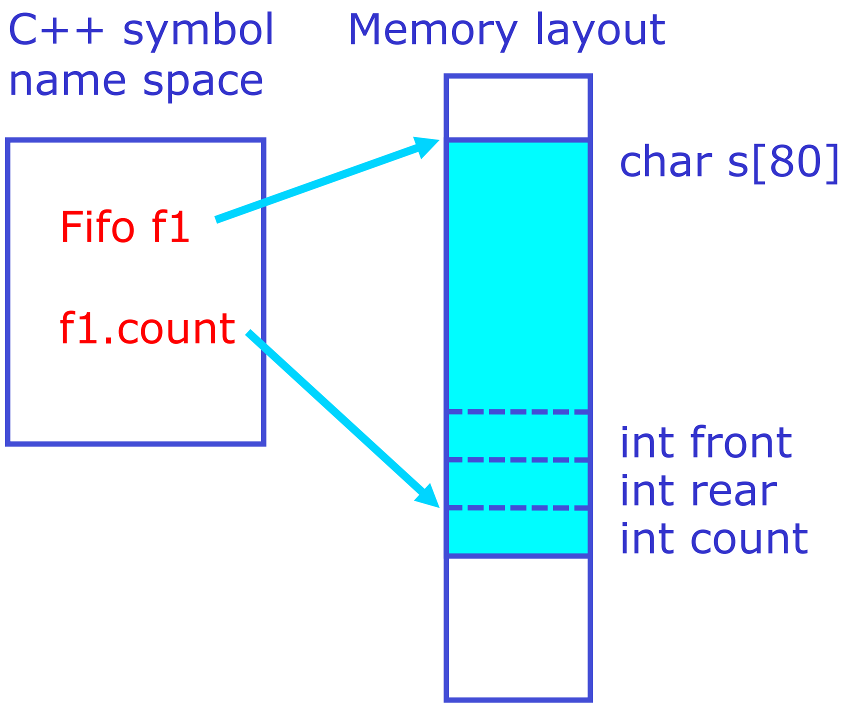
- *Instance* name (`f1,f2`) maps to address in memory
- *Type* name (`Fifo`) controls size of memory allocation, interpretation of memory in allocated block



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

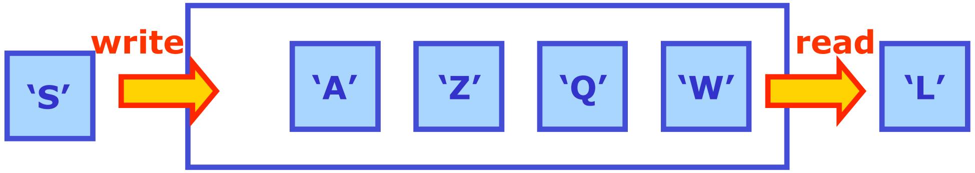
```
const int LEN = 80 ; // default fifo length

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
```

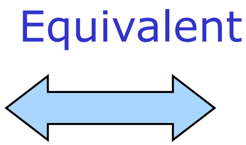
front is an implementation detail that's not part of the abstract FIFO concept. Hiding this detail promotes encapsulation as we are now able to change the implementation later with the certainty that we will not break existing code



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 …

} ;
```



```
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!

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

Common initialization in multiple constructors

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

Not Allowed in C++2003!!!
(Compiler Error)

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

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

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

Common initialization in multiple constructors

- Another clever but wrong solution (for C++2003)
  - Idea: Call `Array(size)` as if it were a regular member function, which will then perform the necessary initialization steps

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

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

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

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

Common initialization in multiple constructors

- The correct solution in C++2003 is to make a private initializer that is called from all relevant constructors

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

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

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

Constructor delegation in C++2011

- New feature of C++2011 is that constructor delegation is explicitly supported – preferred solution

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

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

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

Allowed in C++2011!!!
(New feature)

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

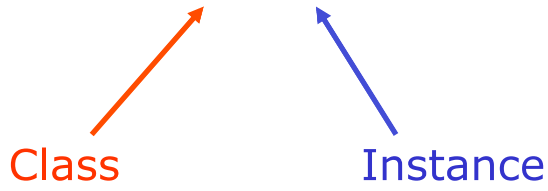
Deletion of automatic variable f calls destructor & closes file automatically

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

Classes vs *Instances* – an important concept

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

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



Class

Instance

- A class can have more than one instance

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

Classes vs *Instances* – an important concept

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

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

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

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

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

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

Intermezzo – Referring to yourself – this

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

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

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

Intermezzo – Referring to yourself – this

- You don’t need *this* very often.
  - If you think you do, think hard if you can avoid it, you usually can

- Most common cases where you really need *this* are
  - Identifying yourself to an outside function (see below)
  - In assignment operations, to check that you’re not copying onto yourself (e.g. a1=a1). We’ll come back to this later

- How to identify yourself to the outside world?
  - Example: Member function of classA needs to call external function externalFunc() that takes reference to classA

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

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

Copy constructor – a special constructor

- The copy constructor is the constructor with the signature

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

- It is used to make a clone of your object

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

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

Taking good care of your property

- Use ‘ownership’ semantics in classes as well
  - Keep track of who is responsible for resources allocated by your object
  - The constructor and destructor of a class allow you to automatically manage your initialization/cleanup
  - All private resources are always owned by the class so make sure that the destructor always releases those
- Be careful what happens to ‘owned’ objects when you make a copy of an object
  - Remember: default copy constructor calls copy ctor on all class data member and copies values of all basic types
  - **Pointers are basic types**
  - If an ‘owned’ pointer is copied by the copy constructor it is no longer clear which instance owns the object → **danger ahead!**

Taking good care of your property

- Example of default copy constructor wreaking havoc

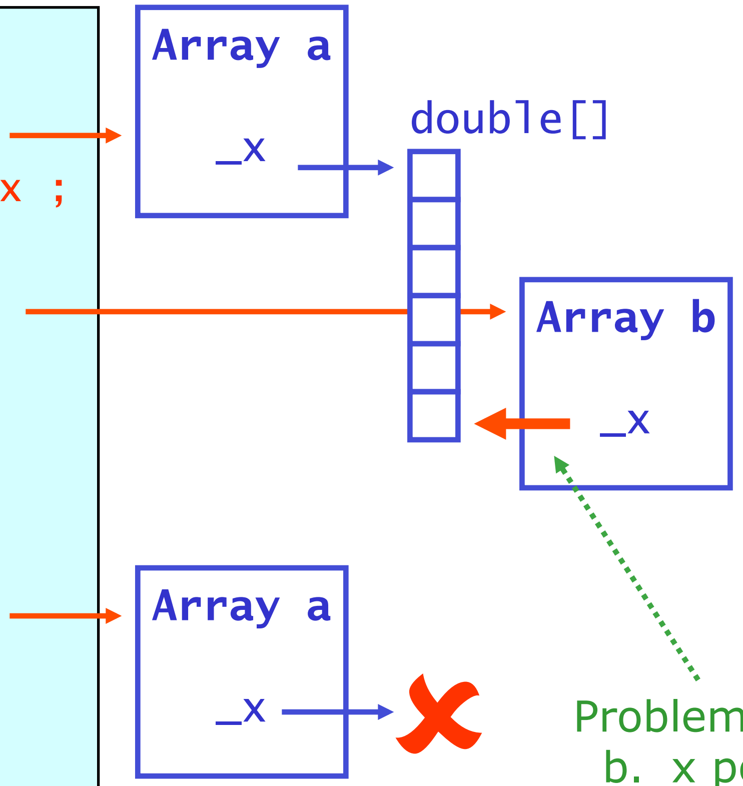
```
class Array {
public:
  Array(int size) {
    initialize(size) ;
  }
  ~Array() {
    delete[] _x ;
  }

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

Watch out! Pointer data member

Taking good care of your property

- 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() ; }
   …
} ;
```

Ownership and defensive programming

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

Const and Objects

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

- Reminder: const and function arguments

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

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

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

Const member functions

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

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

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

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

Const member functions

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

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

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

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

A member function is declared const by putting ‘const’ behind the function declaration

Const member function – the flip side

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

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

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

Const member functions – indecent exposure

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

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

Const return values

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

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

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

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

Why const is good

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

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

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

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

Mutable data members

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

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

  - Use sparingly!

Static data members

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

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

Type ClassName::Name = value ;
```

Declaration

Definition and initialization

Static data members

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

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

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

int Counter::count = 0 ;
```

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

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

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

Static function members

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

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

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

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

```
ClassName::Name(arg,…) ;
```

Static member functions

- Example use case – modification of preceding example

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

int Counter::count = 0 ;
```

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

  Counter c1 ;
  Counter::print() ;

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

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