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

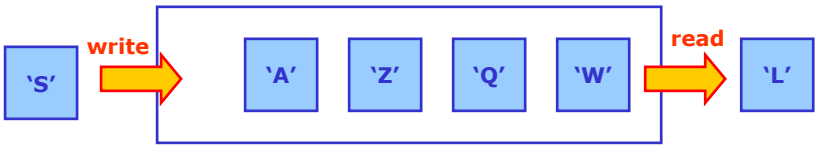
```
Window w ;     // Window object
Database ood ; // Database object
Device d ;     // Device object
```


131

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



132

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

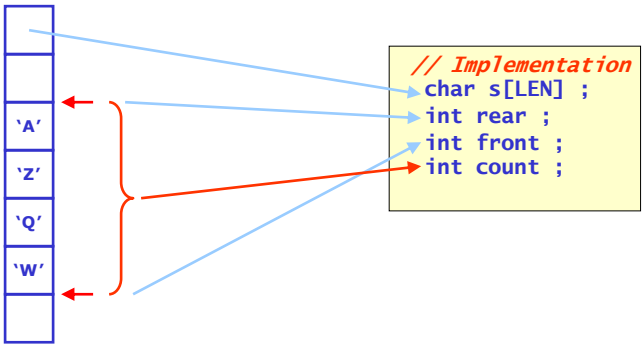
Designing the C++ class FIFO – interface

- 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

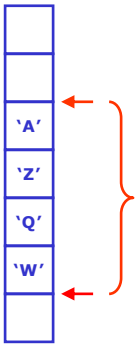
- Implement FIFO with array of elements
  - Use index integers to keep track of front and rear, size of queue



135

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



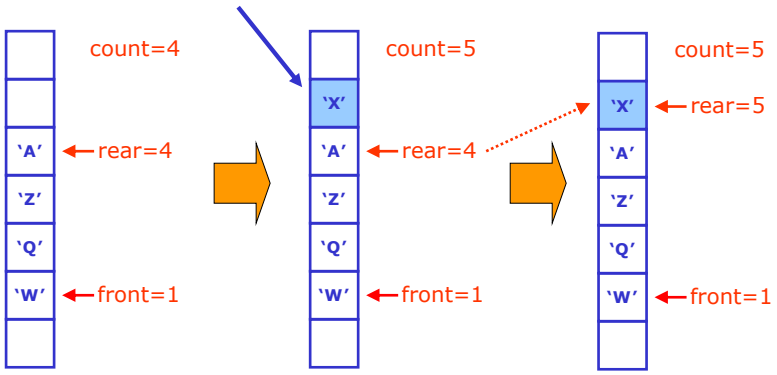

```
// Implementation
void init() { front = rear = count = 0 ; }

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) ; }
```


136

Designing the C++ struct FIFO – implementation

- Animation of FIFO write operation

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

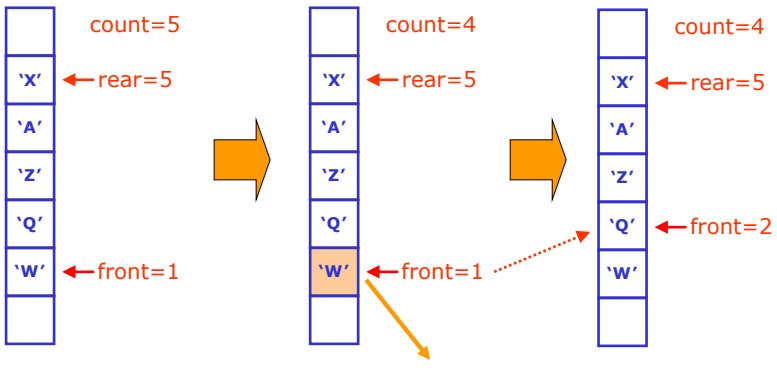


137

Designing the C++ struct FIFO – implementation

- Animation of FIFO read operation

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



138

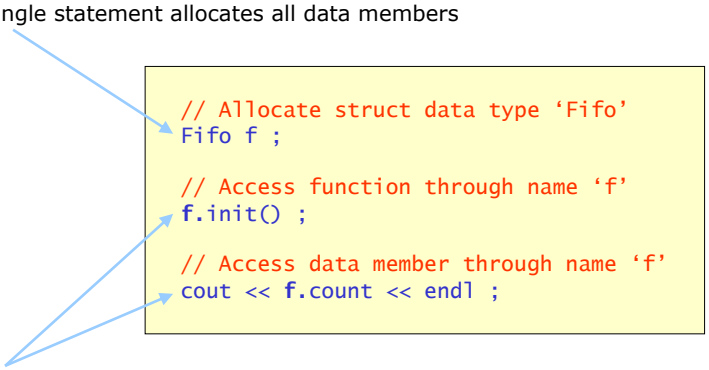
Putting the FIFO together – the struct concept

- 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

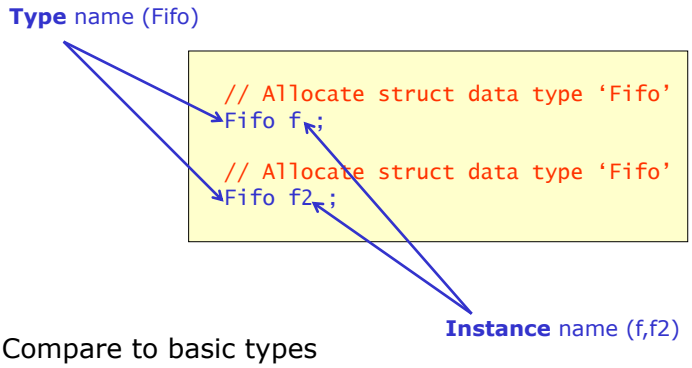


```
// Allocate struct data type 'Fifo'
Fifo f ;

// 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



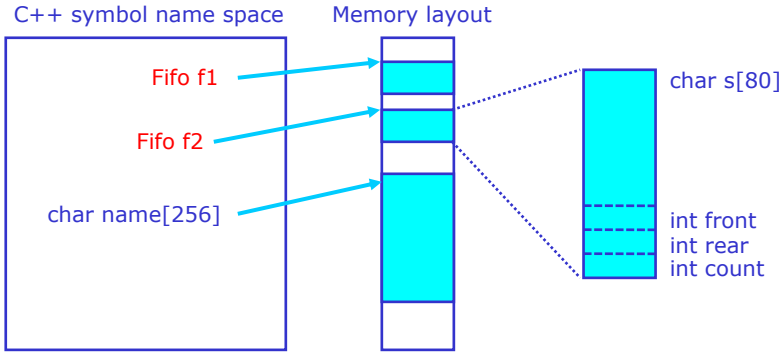
- Compare to basic types

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


141

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

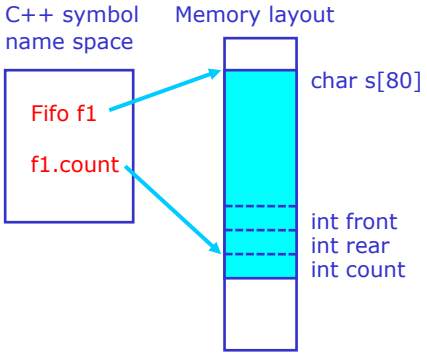


142

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++] ; }
} ;
```


149

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

150

Class – a better struct

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

Equivalent

```
struct Name {
private:

… members …

public:

… members …

} ;
```

```
class Name {


… members …

public:

… members …

} ;
```


Classes and namespaces

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

Declaration and definition

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

Declaration only

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

Definition

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

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

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)

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 ;
};
```