Object-based programming – Classes

3 Class Basics

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

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

174

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!**

175

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 ;
};
```

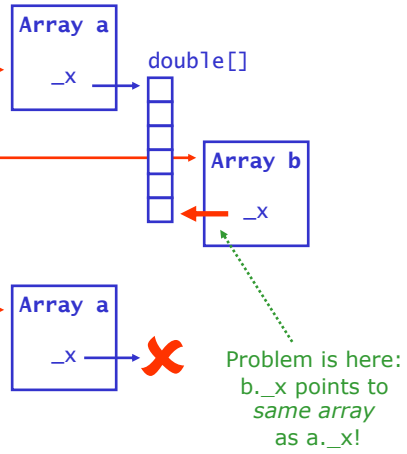
Watch out! Pointer data member

176

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!
}
```



177

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!

178

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

179

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
}
```


186

Const member functions – indecent exposure

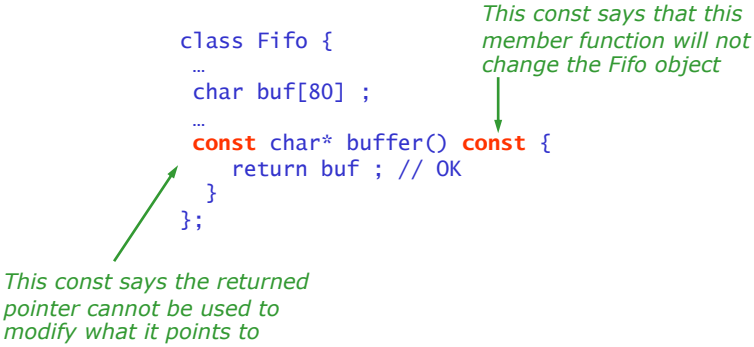
- Const member functions are also enforced not to 'leak' non-const references or pointers that allows users to change its content

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


187

Const return values

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



188

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

189

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
```

192

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,…) ;
```


193

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
```