C++ Programming

~ Object-based Programming ~

Prepared for Ingenix, Inc.

FRESH SOURCES

Object-based Programming

- Combines related data and functions
- Originated with Simula-67
- Objects are instances of “classes”
  - Structures with member functions
- Aids greatly in program organization
  - The class is the basic unit of modularity
  - Some objects have a unique identity
  - Objects can contain other objects

Agenda

- Member functions
- Constructors
- Member access control
- Destructors
- Object Management
- Dynamic Objects
- Operator Overloading
- Using IOStreams
- Static Members
- Volume 1: 4-6, 11-13; Volume 2: 4

Member Functions

- Called on behalf of an object of the class type
- Uses “dot” syntax:
  `anObject.f()`
- Inside member functions, the keyword “this” gives a pointer to the object in the call
- Example: employee2.cpp

Constructors

- Called automatically when objects are created
  - To initialize them
  - They do not acquire storage for them
- The initializer list is used to transfer data from constructor parameters to data members
  - *Should* be used for objects (efficiency)
  - Optional for built-ins
  - *Must* be used for special members
    - **const**, reference members (rarely used)
  - Initialized in *declaration order!*

Access Control

- Data members should not be modifiable by users
  - Objects should control their innards!
  - Users only should see an *interface*
- Data can be made "private"
  - So can functions (hidden implementation details)
  - Using the **class** keyword instead of struct makes things private
  - Use **public** where needed
- Example: employee3.cpp

const Member Functions

- It is allowed and often useful to make objects **const**
  - They must be initialized when declared, of course
- No member function should change a **const** object's state
  - From the user's point of view
- You must declare which functions are safe to call for **const** objects
- Use it *everywhere* it applies!!!
- Example: employee4.cpp

The Default Constructor

- A constructor that takes no arguments
- Not always needed
  - Don’t blindly define one!
  - Our Employee class doesn’t really need one
- Compiler defines it for you if you define *no constructors at all*
- The compiler-generated default constructor doesn’t really do anything
  - Member sub-objects are always initialized anyway
  - It is generated simply to let you define empty objects
- Example: time.cpp

Object Initialization

- All data members with constructors are initialized automatically
  - In declaration order
  - If they appear in the initializer list, they use the appropriate constructor
  - If they don't appear there, then their default constructor is used (must exist in that case!)
- Then the body of the matching constructor executes
- Examples: badInit.cpp, goodInit.cpp

Destructors

- Called automatically when an object "dies"
  - Goes out of scope, or explicitly deleted, say
- Use to "de-initialize" an object
  - "~ClassName( )" syntax
- Most objects don't need one!
  - Only if they manage internal resources (memory, files, connections, etc.)
  - A *pointer member* is a hint that a destructor is probably needed
- Example: File.cpp

A String Class

```
#include <cstring>
#include <iostream>

class String
{
    char* data;

public:
    String(const char* s = "")
    {
        data = new char[std::strlen(s) + 1];
        std::strcpy(data,s);
    }
    ~String() {delete [] data;}   // Destructor
    int size() const {return std::strlen(data);}
    char getAt(int pos) const {return data[pos];}
    void setAt(int pos, char c) {data[pos] = c;}
    void display()
    {
        std::cout << data;
    }
};
```

Using class String

```
int main()
{
    String s = "hello"; // same as String s("hello");
    for (int i = 0; i < s.size(); ++i)
        cout << "s[" << i << "] == "
             << s.getAt(i) << std::endl;

    String empty;
    std::cout << '"';
    empty.display();
    std::cout << "\"\n";
}

/* Output:
s[0] == h
s[1] == e
s[2] == l
s[3] == l
s[4] == o
""
*/
```

Strange Behavior

```
int main()
{
    String s = "hello";
    String t = s;          // same as String t(s);
    t.setAt(0,'j');
    s.display();
}

/* Output:
jello
<The instruction at "0x004022dd" referenced memory at
"0x00000008". The memory could not be "written".
*/
```

Initialization vs. Assignment

- Initialization occurs only *once*, when an object is *created*
  - always by some *constructor*
- Assignment occurs only *after* an object has been initialized
  - via `operator=`
- What constructor executed in the previous slide?

The Copy Constructor

- Initializes a new object as a copy of an existing object
  - of the same type
- Has signature `T::T(const T&)`
- Copies each member across
  - using their own copy constructors recursively
- Generated by compiler
  - But you can override it (and sometimes should)

Compiler-generated Copy Ctor

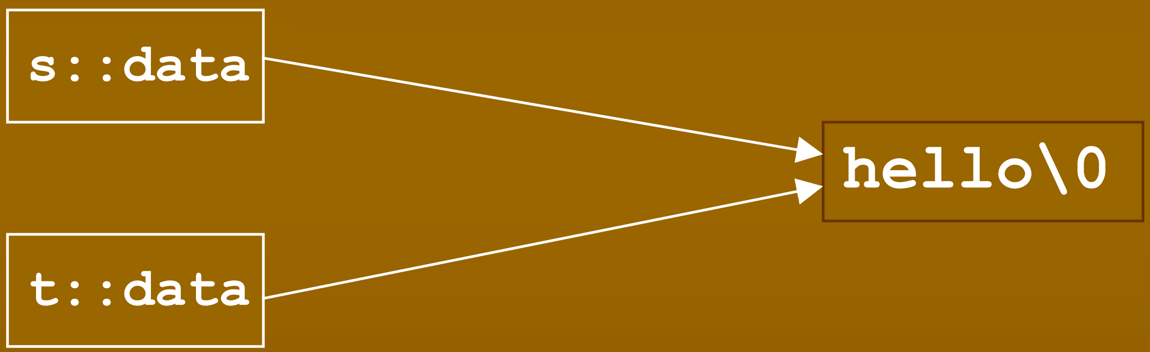
```
String(const String& s)
    : data(s.data)
{}

// Identical to (in our case):

String(const String& s)
{
    data = s.data;
}

// because pointers are not objects.
```

“Shallow Copy”



Problems with Shallow Copy

- If you have a *pointer* as a data member, a shallow copy is probably not what you want
- By changing the referent in one object, you also change it in the other object
- If you de-allocate the data member in one object, you have created a likely fatal situation in the other (double delete)

What should the Copy Ctor Do?

- Make a “Deep Copy”:
  - Allocate new heap space
  - Copy characters to target

A "Deep Copy" Copy Constructor

```
// You must do this when you need deep copy:
String(const String& s)
{
    data = new char[strlen(s.data)+1];
    strcpy(data, s.data);
}
```

More Strange Behavior

- Why does changing `t` affect `s` below?

```
int main()
{
    String s = "hello"; // same as String s("hello");
    String t;
    t = s;
    t[0] = 'j';
    cout << s << endl;
}
```

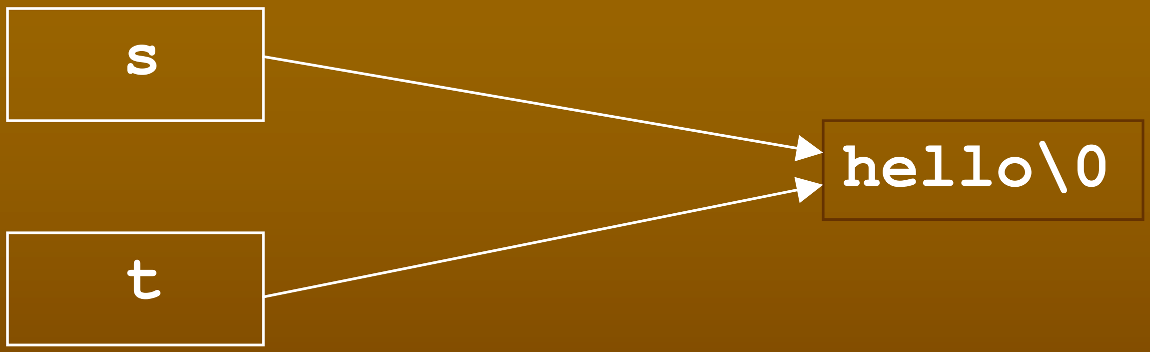
jello

Object Assignment

- Uses **`operator=`**
  - must be a member
- Generated by the compiler
  - assigns individual members
  - **`String::operator=`** just assigns the pointer **`data`**
  - but we want to *replicate* the underlying characters!
- You can override it
  - and should whenever an object's state is external to itself
  - a pointer (or reference) member is a sign that **`operator=`** needs help

Compiler-generated Assignment

```
String& String::operator=(const String& rhs)
{
    data = rhs.data;
    return *this;
}
```



What should `String::operator=` Do?

- Allocate new heap space
- Copy characters to target
- Delete the old heap space
- Return `*this`
- Avoid Self-assignment
  - For optimization

Correct Assignment

```
String& String::operator=(const String& s)
{
    if (&s != this)
    {
        char* new_data = new char[strlen(s.data)+1];
        strcpy(new_data, s.data);
        delete [] data;
        data = new_data;
    }
    return *this;
}
```

Object Management Summary

- Copy Constructor
  - the compiler generates it only if you don’t
  - does shallow copy
- All Other Constructors
  - if you don’t provide any constructors at all, the compiler generates a *default constructor* (which default-constructs each member)
  - Single-arg constructors are special (“conversion constructors”)
- Assignment Operator
  - the compiler generates it only if you don’t
  - does shallow assignment
- Destructor
  - the compiler generates only if you don’t (calls each member’s destructor)

Standard Conversions

- *Implicit* promotion of numeric types
- Widening of:
  - char -> short -> int -> long
- Promotion from integer to floating-point
- Occurs in mixed-mode expressions (x + i) and in passing parameters
  - Prototypes initiate the conversion for parms

Implicit Conversions to Class Type

- Achieved through a single-argument constructor
  - Also called a “converting constructor”
  - Or less commonly, through a conversion operator (will see later)
- You can turn it off
  - With a special keyword (**explicit**)
- Example: convert.cpp

Dynamic Objects

- Can store objects on the heap (“free store”)
- Use the **new** operator
- You get a pointer back
- Objects are always initialized
  - A constructor executes
  - You can also initialize built-ins in the new statement:
    ```
    int* p = new int(8);      // *p == 8
    ```
- Always **delete** a heap pointer when you’re finished
  - The biggest source of C++ bugs!
- Example: initObjects.cpp

Operator Overloading

- Binary operators
- Unary operators
- Stream operators
- Conversion Operators
- Special operators (may skip some of these)
  - Indexing
  - Pre-post increment/decrement
  - Smart pointers (operator ->)
  - Function call (crucial to STL)

Why Operator Overloading?

- A Notational Convenience
- Motivated by Mathematics
  - e.g., basic operations on matrices, vectors, etc.
- Results in clearer code
  - if used wisely!

Complex Numbers

- The Canonical Example
- Complex numbers are pairs of real numbers
  - (real, imaginary), e.g., (2,3), (1.04, -12.4)
  - like points in the x-y plane
  - (a, b) + (c, d) = (a+c, b+d)
  - applications in Electrical Engineering
- Compare function-style operations to using operator functions

A `complex` Class

```
#include <iostream>

class complex
{
    double real;
    double imag;

public:
    complex(double real=0, double imag=0)
    {
        this->real = real;
        this->imag = imag;
    }
    complex add(const complex& c) const
    {
        complex result;
        result.real = this->real + c.real;
        result.imag = this->imag + c.imag;
        return result;
    }
    void display(std::ostream& os) const
    {
        os << '(' << real << ',' << imag << ')';
    }
};
```

Using the `complex` Class

```
using namespace std;

int main()
{
    complex z(1,2), w(3,4);
    complex a = z.add(w);      // want a = z+w
    a.display(cout);
}

(4,6)
```

Operator Functions

- **`operator`** keyword together with an operator
  - **`operator+`**
  - **`operator-`**
  - etc.
- Can overload all operators except:
  - **`::`**
  - **`.`**
  - **`.*`**
  - **`?:`**

complex with operator+

```
class complex
{
    // ...
public:
    // ...
    complex operator+(const complex& c) const
    {
        complex result;
        result.real = this->real + c.real;
        result.imag = this->imag + c.imag;
        return result;
    }
    // ...
};

int main()
{
    complex z(1,2), w(3,4);
    complex a = z + w;              // z.operator+(w)
    a.display(cout);
}
```

Rules

- **`operator+`** is the function name
  - you can also invoke it directly as **`z.operator+(w)`**
  - a *binary* function, of course
- Normal precedence rules apply
- Can be either a global or member function
- If member, **`this`** is the left operand
- If global, one argument must be user-defined

Conversion Constructors and Operator Overloading

- For example `T::T(int)`
- Can use 2 initialization syntaxes:
  - `T t(1);`
  - `T t = 1;`
- Provide implicit conversions
  - `t + 1` becomes `t + T(1)`
- Can disable with the `explicit` keyword

`complex` Conversion

```
class complex
{
public:
    complex(double real = 0, double imag = 0)
    {
        this->real = real;
        this->imag = imag;
    }
    // …
};

int main()
{
    complex w(3,4);
    complex a = w + 1;                  // w + (1,0)
    a.display(cout);

}

(4,4)
```

Member vs. Non-member Operators

- The expression `1 + w` does not compile.
- Left-operand must be a class object to match a member operator
- A global operator will apply an implicit conversion to *either* operand via a single-arg constructor, if available
- In general, *binary* operators should be *global*
- In general, *unary* operators should be *members*

Global `complex` Operators

```
class complex
{
public:
    complex(double real = 0, double imag = 0)
    {
        this->real = real;
        this->imag = imag;
    }
    double getReal() const {return real;}
    double getImag() const {return imag;}
    // ...
};

complex operator+(const complex& c1, const complex& c2)
{
    double real = c1.getReal() + c2.getReal();
    double imag = c1.getImag() + c2.getImag();
    complex result(real, imag);
    return result;
}
```

A Unary `complex` Operator

```
class complex
{
public:
    complex operator-() const
    {
        return complex(-real, -imag);
    }
    // …
};

int main()
{
    complex w(3,4);
    complex a = -w;
    a.display(cout);
}

(-3,-4)
```

Stream Operators

- **`operator <<`** and **`operator >>`**
- Must be global
  - because left operand is a stream!
- Stream is passed by reference
  - for efficiency
- Should return the stream
  - to support chaining insertions and extractions

`complex` Stream Output

```
ostream& operator<<(ostream& os, const complex& c)
{
    os << '(' << c.getReal() << ','
       << c.getImag() << ')';
    return os;
}

int main()
{
    complex w(3,4);
    complex a = -w;
    cout << a << endl;
}
```

(-3,-4)

A “Complete” `complex`

- Would provide all pertinent operators
  - including assignment ops such as +=, -=, etc.
  - assignment ops must be *members*
- Provides stream insertion and extraction
- Global functions are class friends
  - not necessary, just a convenience

`operator[]`

- A Unary Operator
- Must be a member
- For “array-like” things
  - vectors, strings
- Must provide two versions:
  - version for const objects
  - version for non-const objects
  - other operators may require this special handling

A “Safe” Array class

```
class Index
{
    enum {N = 100};
    int data[N];
    int size;

public:
    Index(int n)
    {
        if (n > N)
            throw "dimension error";
        for (int i = 0; i < n; ++i)
            data[i] = i;
        size = n;
    }
    int getSize() const {return size;}
    int& operator[](int i)
    {
        if (i < 0 || i >= size)
            throw "index error";
        return data[i];
    }
};
```

Using Index

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

int main()
{
    Index a(10);
    for (int i = 0; i < a.getSize(); ++i)
        cout << a[i] << ' ';
    cout << endl;
    a[5] = 99;
    cout << a[5] << endl;
    cout << a[10] << endl;
}
```

```
0 1 2 3 4 5 6 7 8 9
99
abnormal program termination
```

Using a `const` Index

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

int main()
{
    const Index a(10);              // a const Index
    for (int i = 0; i < a.getSize(); ++i)
        cout << a[i] << ' ';        // COMPILE ERROR!
    cout << endl;
}
```

Supporting a `const` Index

```
class Index
{
    // ...
    int& operator[](int i)
    {
        if (i < 0 || i >= size)
            throw "index error";
        return data[i];
    }

    int operator[](int i) const     // A const version
    {
        if (i < 0 || i >= size)
            throw "index error";
        return data[i];
    }
};
```

Conversion Operators

- The complement to single-arg constructors
- Provide implicit conversions *to* another type
- Member function with the signature:

```
operator T() const;
```

Index-to-double Conversion

```
class Index
{
    // ...
    operator double() const
    {
        double sum = data[0];
        for (int i = 1; i < size; ++i)
            sum += data[i];
        return sum / size;
    }
};

int main()
{
    const Index a(10);
    double x = a + 1;
    cout << x << endl;
}

5.5
```

Warning!

- Why *shouldn't* the **`complex`** class have a conversion operator to **`int`** or **`double`**?
- Hint: consider the expression

  ```
  w + 1
  ```

  where **`w`** is **`complex`**.
- You can turn off implicit conversions of single-arg constructors with the **`explicit`** keyword:

  explicit complex(double = 0, double = 0);

Other Operators

- -> for "smart pointers"
  - e.g., auto_ptr in the standard library
- ++, --
  - Pre and post versions
- ( ), "function-call" operator
  - We'll see this when we do STL

Overloading operator->

- For when you want to add functionality to the built-in operator->
- Must return a raw pointer
  - Or something that can be dereferenced
- Example: SafePtr.cpp

auto_ptr

- A standard wrapper for memory allocation
- A smart pointer
  - Can use -> and * normally
- Its destructor automatically calls **delete**
  - Which automatically calls the destructor
  - Unfortunately, can’t use for arrays
- Example: File2.cpp

Overloading ++ and --

- Must distinguish between pre and post
  - Post versions take an extraneous int argument
- The post versions must save current value
  - That's why the pre versions are more efficient
  - They should also return a const object
    - To disallow x++++
      - Illegal, modifies temporary
- Examples: PrePost.cpp, SafeArrayPtr.cpp

Overloading operator()

- The Function Call Operator
- Constitutes a Function Object
  - An object that can behave like a function
  - Compensates for C++ not being Lisp!
- If class T::operator() exists:
  - Then t( ) acts like a function call
- Example: findGreater.cpp, findGreater2.cpp

Using IOStreams

- IOStreams are powerful objects
- Creating input operators are tricky
  - Also called “extractors”
  - operator>>
- Must set stream state

Why IOStreams?

- Brings the advantages of objects to I/O
- Constructors connect to sources/sinks
- Destructors disconnect
- Can get/set stream state
- Operator Overloading

Inserters

- Inserts an object into a stream
  - that is, it does output
- Uses operator<<
  - the left-shift operator
  - the arrow suggest the direction of the data flow
- Easy to define for your own classes

Defining an Inserter

- The signature of the function is:
  ostream& operator<<(ostream&, const T&);
- The stream is *not* const
  - because its state will change
- You return a reference to the stream
  - to allow chaining (multiple “<<”s)

Inserter Example

- A Date class inserter:

```
ostream& operator<<(ostream& os, const Date& d) {
  char fillc = os.fill('0');
  os << setw(2) << d.getMonth() << '-'
     << setw(2) << d.getDay() << '-'
     << setw(4) << setfill(fillc) << d.getYear();
  return os;
}
```

Extractors

- Consume input from a stream
- Uses operator>>
- Signature is
  istream& operator>>(istream&, T&);
- The object is not const because it is going to get overwritten with input!
- How do you know if it worked
  - e.g., you want an int and got alphas

Extractor Example

- A Date class again:

```
istream& operator>>(istream& is, Date& d) {
  is >> d.month;
  char dash;
  is >> dash;
  if(dash != '-')
    is.setstate(ios::failbit);
  is >> d.day;
  is >> dash;
  if(dash != '-')
    is.setstate(ios::failbit);
  is >> d.year;
  return is;
}
```

Stream State

- 4 states:
  - good
  - eof
  - fail (unexpected input type, like alpha for numeric)
    - also set by eof
  - bad (device failure)
- Once a stream is no longer good, you can’t use it
  - all ops are no-ops
  - Can clear with clear( ) (must after a failed op!)
- Can test with associated functions:
  - good( ), eof( ), fail( ), bad( )
- Can test for good( ) like this:
  - if (theStream) [same as if (theStream.good( ))]

Streams and Exceptions

- Can have exceptions thrown instead of checking state
- Call the exceptions( ) member function
- Can pick which states you want to throw: myStream.exceptions(ios::badbit);
- The exception type thrown is ios::failure
  - we'll see the **ios** base class later

File Streams

- Classes ifstream, ofstream, fstream
  - declared in <fstream>
- Constructors open, destructors close
- All normal stream operations apply
- Additional member functions:
  - close( ), open( )
- Open modes
  - ios::in, ios::out, in::app, in::ate, ios::trunc, ios::binary
  - Can combine with the bitwise or (|)
- Example: StrFile.cpp

String Streams

- Classes istringstream, ostringstream, stringstream
  - declared in <sstream>
- Writes to or reads from a string
  - or both
- Useful for converting other data types to and from strings
- Examples: IString.cpp, DateIOTest.cpp, Ostring.cpp

Output Formatting

- Can set stream attributes
  - width, fill character, alignment, numeric base, floating-point format, decimal precision, etc.
- Use setf( ) and unsetf( )
- Example: Format.cpp

Object "Serial Numbers"

- Suppose you want to have an object id field that automatically increments whenever you create an object
- Where do you store that counter?

Static Data Members

- Belong to the whole class
  - Not each object
- Have static storage class
  - Just like globals and local statics
- But are inside the scope of their class
- Must *declare inside* the class, but *define outside* the class!
- Example: serialObjects.cpp

Counting Objects

- Similar to the serial number issue
- Except keeps a *current* count
  - Destructor decrements counter
- Question: How do you retrieve the count through a method
  - Remember, public data is *bad*.

Static Member Functions

- "Class Methods"
- Free-standing functions (like globals)
- But are in the scope of the class
- Have no "this" pointer
  - Are called without an object
  - T::f( );
- Example: countObjects.cpp

Class Constants

- Static members that are **const**
- Can initialize *inside* class, if desired
  - Can use as an array dimension, for example
  - You *still* have to define the space *outside* the class definition
- Another technique:
  - Using **enum** in a class (I prefer it)

Static Class Constants

```
class Customer
{
private:
    static const int MAXCONTACTS = 100;
    Contact contacts[MAXCONTACTS];
…
};

const int Customer::MAXCONTACTS; // No initializer!
```

Enumerated Constants

- Uses enum keyword
- Defaults to 0, 1, 2, …
- True compile-time, integral constants
  - Take no space
  - Therefore, behave like statics
    - Don't occupy space in an object

enum Example

```
class Customer
{
private:
    enum {MAXCONTACTS = 100};
    Contact contacts[MAXCONTACTS];
…
};

// No definition here!
```

Exercise

- Create a class, **Rational**, that supports rational numbers (fractions) as explained in Rational.doc
- This is time consuming but worth it.
- Try to do it stepwise:
  - Implement some of the functionality, then test
  - Repeat
  - Full test program in trational.cpp