Fast Introduction to Object Oriented Programming and C++

Daniel G. Aliaga

Note: a compilation of slides from Jacques de Wet, Ohio State University, Chad Willwerth, and Daniel Aliaga.

Outline

- Programming and C++
- C vs. C++
- Input/Output Library
- Object Classes
- Function and Operator Overloading
- Inheritance and Virtual Functions

Outline

- Programming and C++
- C vs. C++
- Input/Output Library
- Object Classes
- Function and Operator Overloading
- Inheritance and Virtual Functions

Object Oriented Programming (OOP)

- OOP models communication between objects
  - just as people send messages to one another, objects communicate via messages.
- OOP encapsulates data (attributes) and functions (behavior) into objects.
  - Objects have *information hiding*.

Object Oriented Programming (OOP) cont.

- Objects are *user-defined types* called *classes*.
- A class definition is an extension of a C struct.
- The variables are typically private or protected and the functions are typically public.
- structs and classes are the *same in C++* (*both* can have member functions or “methods”), but struct members are public *by default* and class members are private *by default*.

C++

- Development of C in early ’70s
- Early ’80s, new language by Bjarne Stroustrup ⇒ “C with Classes”
- October 1985 - first commercial release
- November 1997 – ANSI Standard for C++
- C++ is the most popular language now
  - Portability, flexibility, code reuse and quality

C++

Is C++ the best language? (Better than C?)

- Offers some nice features
- Choice of language though is a result of the particular problem and the environment
  - Nevertheless, bad programming results a bad program regardless of the chosen language
- Not only a new language, but a new way of thinking
- Additional support for programming, software engineering, and large projects

C++

Claimed advantages over C

1. Faster development time (code reuse)
2. Creating/using new data types is easier
3. Memory management easier ➔ less leaks
4. Stricter syntax & type checking ➔ less bugs
5. Data hiding easier to implement
6. Object-oriented (OO) concepts in C++ allows direct coding from an OO Analysis and Design document

Outline

- Programming and C++
- C vs. C++
- Input/Output Library
- Object Classes
- Function and Operator Overloading
- Inheritance and Virtual Functions

C versus C++

Differences:

- C++ is a superset of C
- Namespaces
- Variable declaration
- NULL-pointers vs. 0-pointers
- Stricter type checking

C versus C++

Differences:

- Bool data type: value of ‘true’ or ‘false’
- Const vs. non-const
  - e.g., int i = (int)12.45;
  - e.g., const char *name = “Daniel”;
  - e.g., const char *p = name;

C versus C++

Differences:

- ‘void’ parameter list
  - void function() and void function(void)
- Using C functions in a C++ application
  - extern “C”
- Function overloading is possible in C++
  - Same name, different parameter list

C versus C++

Differences:

- Default function arguments
- typedef – still allowed but not necessary
- Functions as part of a structure
- Function return values
  - In C, default is int
  - In C++ you have to specify the value, otherwise the compiler will give a warning.

C versus C++

```
// This is a comment that covers
//  more than one line.  It
//  requires double slashes on
//  each line.
```

C versus C++

inline float **equation** (float **x**)
        { return **x*x-5x+3**; }

// This solves problems that arise with

#define **equation**(**x**) **(x*x)-5x+3**

C versus C++

Differences:

- “new” instead of “malloc()”
- “delete instead of “free()”
  - Example: “int *values = new values[10];”
  - Example: “delete [] values;”

Outline

- Programming and C++
- C vs. C++
- <u>Input/Output Library</u>
- Object Classes
- Function and Operator Overloading
- Inheritance and Virtual Functions

Input/Output Library in C++

- It is perfectly valid to use the same I/O statements in C++ as in C -- The very same *printf*, *scanf*, and other *stdio.h* functions that have been used until now.

- However, C++ provides an alternative with the new stream input/output features. The header file is named iostream and the stream I/O capabilities are accessible when you use the pre-processor declaration:

```
#include <iostream>       // No ".h" on std headers
using namespace std;      // To avoid things like
                          // std::cout and std::cin
```

Input/Output Library in C++

- Several new I/O objects available when you include the iostream header file. Two important ones are:
  - cin // Used for keyboard input (std::cin)
  - cout // Used for screen output (std::cout)
- Both *cin* and *cout* can be combined with other member functions for a wide variety of special I/O capabilities in program applications.

Input/Output Library in C++

- Since cin and cout are C++ objects, they are somewhat "intelligent":
  - They do not require the usual format strings and conversion specifications.
  - They do automatically know what data types are involved.
  - They do not need the address operator, &.
  - They do require the use of the stream extraction (>>) and insertion (<<) operators.
- The next slide shows an example of the use of cin and cout.

Example using cin and cout

```
#include <iostream>
using namespace std;        // replace every cin and cout
                            // with std::cin and std::cout
                            // without this line
int main ( )
{
   int a, b;   float k;   char name[30];
   cout << "Enter your name\n" ;
   cin  >>  name ;
   cout  <<  "Enter two integers and a float\n" ;
   cin  >>  a  >>  b  >>  k ;
   cout << "Thank you, " << name << ", you entered\n " ;
   cout  <<  a  <<  ", "  <<  b << ", and "  <<  k  <<  '\n' ;
}
```

Example Program Output

Enter your name

Rick

Enter two integers and a float

20 30 45.67

Thank you, Rick, you entered

20, 30, and 45.67

Input Stream Object Member Functions

cin.getline (**array_name, max_size**) ;

Example:

```
char name[40] ;
cin.getline (name, 40); // gets a string from
                        // keyboard and assigns
                        // to name
```

Outline

- Programming and C++
- C vs. C++
- Input/Output Library
- <u>Object Classes</u>
- Function and Operator Overloading
- Inheritance and Virtual Functions

Object Classes in C++

- Classes enable a C++ program to model *objects* that have:
  - attributes (represented by *data members*).
  - behaviors or operations (represented by *member functions).*

- Types containing *data members* and *member function* prototypes are normally defined in a C++ program by using the keyword *class.*

Object Classes in C++

- A class definition begins with the keyword *class*.
- The body of the class is contained within a set of braces, { } ; (notice the semi-colon).
- Within the body, the keywords *private:* and *public:* specify the access level of the members of the class. Classes default to private.
- Usually, the data members of a class are declared in the *private:* section of the class and the member functions are in *public:* section.
- Private members of the class are normally not accessible outside the class, i.e., the information is hidden from "clients" outside the class.

Object Classes in C++

- A member function prototype which has the very same name as the name of the class may be specified and is called the constructor function.

- The definition of each member function is "tied" back to the class by using the binary scope resolution operator (::).

- The operators used to access class members are identical to the operators used to access structure members, e.g., the dot operator (.).

Classes Example

```
#include <iostream>
#include <cstring>          // This is the same as string.h in C
using namespace std;

class Numbers // Class definition
{
   public:                  // Can be accessed by a "client".
      Numbers ( ) ;         // Class "constructor"
      void display ( ) ;
      void update ( ) ;
   private:                 // Cannot be accessed by "client"
      char name[30] ;
      int a ;
      float b ;
} ;
```

Classes Example (continued)

```
Numbers::Numbers ( ) // Constructor member function
{
   strcpy (name, "Unknown") ;
   a = 0;
   b = 0.0;
}

void Numbers::display ( )           // Member function
{
   cout  <<  "\nThe name is "  << name  <<  "\n" ;
   cout  <<  "The numbers are "  <<  a  <<  " and "  << b
               <<  endl ;
}
```

Classes Example (continued)

```
void Numbers::update ( )                    // Member function
{
   cout  <<  "Enter name"  <<  endl ;
   cin.getline (name, 30) ;
   cout  <<  "Enter a and b"  <<  endl ;
   cin  >>  a  >>  b;
}
```

Classes Example (continued)

```
int main ( )                       // Main program
{
   Numbers no1, no2 ;              // Create two objects of
                                   // the class "Numbers"

   no1.update ( ) ;                // Update the values of
                                   // the data members

   no1.display ( ) ;               // Display the current
   no2.display ( ) ;               // values of the objects
}
```

Example Program Output

```
Enter name
   Rick Freuler
Enter a and b
   9876       5.4321
The name is Rick Freuler
The numbers are 9876 and 5.4321
The name is Unknown
The numbers are 0 and 0
```

More Detailed Classes Example

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

class Numbers                    // Class definition
{
   public:
      Numbers (char [ ] = "Unknown", int = 0, float = 0.0) ;
      void display ( ) ;
      void update ( ) ;
   private:
      char name[30];
      int a;
      float b;
} ;
```

More Detailed Classes Example (continued)

```
Numbers::Numbers (char nm[ ], int j, float k )
{
   strcpy (name, nm) ;
   a = j ;
   b = k ;
}

void Numbers::update ( )
{
   cout  <<  "Enter a and b"  <<  endl ;
   cin  >>  a  >>  b ;
}
```

More Detailed Classes Example (continued)

```
void Numbers::display( )
{
   cout  <<  "\nThe name is "  <<  name  <<  '\n' ;
   cout  <<  "The numbers are " <<  a <<  " and " <<  b
              << endl ;
}
int main ( )
{
   Numbers no1, no2 ("John Demel", 12345, 678.9);
   no1.display ( ) ;
   no2.display ( ) ;
}
```

More Detailed Example Program Output

The name is Unknown

The numbers are 0 and 0

The name is John Demel

The numbers are 12345 and 678.9

Outline

- Programming and C++
- C vs. C++
- Input/Output Library
- Object Classes
- <u>Function and Operator Overloading</u>
- Inheritance and Virtual Functions

Function Overloading

- Same function name, different signatures
  - Does not include return type
    - Return type is not considered because it doesn't have to be used
- Example:
  - void Print(int n);
  - void Print(char c);
  - size_t Print(char* str);
  - void Print(float f, int precision);

Name Decoration

- Compiler creates unique name for each function
  - Consists of function name, parameter number and types
  - Called *name decoration* or *name mangling*
  - Visible in “map file” created by linker

Overloading with Pointers

- Pointers to different types are distinct types
  - Works fine, e.g.:
    - void findData(char *s);
    - void findData(int *i);
- Array ⇔ Pointer
  - Array notation and pointer notation is same function
    - int largest(int* values, int count);
    - int largest(int values[], int count);

Overloading with References

- Changing parameter to reference only changes *how* function is called
  - int func(int a);
  - int func(int& a);   // ambiguous function
- Non-const reference parameters may not call correct function
  - Function could change original value
  - Compiler will avoid it if possible
  - Solution: make reference parameters const
    - int func(const int a);   // make sense?
    - int func(const int &a); // make sense?
      - No?
      - Actually it does – for efficiency reasons: data not copied but referred too…

Overloading and Const

long larger(long a, long b);

long larger(const long a, const long b);

- What’s the difference?
  - Both are pass-by-value
  - Original value can’t be changed by either
  - No difference
- Const is only used for differences on pointers and references
  - Controls possible changes to data

Operator Overloading

```
Box box1(10, 15, 20);
Box box2(10, 15, 25);
if( box1 < box2 )
    // do something
```

- Nothing more than a fancy syntax
  - “syntactic sugar”

Why Overload Operators?

- Allows defining a function to be called when an pre-defined operator is found
  - Cannot create your own operators
  - Cannot overload all operators
- A great feature for making algorithms easier to read
  - But don’t go overboard

Operators to Overload

- All built-ins except
  - Scope resolution ::
  - Conditional ?:
  - Member access .
  - Dereference to class member .*
  - Sizeof sizeof
- Can even overload
  - new and delete
  - stream insertion/extraction
  - type conversion (explicit casts)

Introduction to Operator Overloading

```
class Box {
public:
    bool operator < (const Box& RHS) const;
    int volume( void ) const { return length * width * height; }
    …
};
```

- Return type is obvious here
- LHS object is “this” object
- Parameter is RHS object
- Function doesn’t alter this object, so function is const

Implementing Overloaded Operator

```
bool Box::operator < (const Box& RHS) const
{
    return volume() < RHS.volume();
}
```

- Note implied “this” pointer on first “volume()”
- Return statement calls 2 functions, compares values and returns results
- Usage: if(box1 < box2) …
  - Same as: if(box1.operator<(box2))

Shortcuts for Relational Operators

- Given equality (==) and less-than (<) operators, all other relational operators can be defined
  - != ⇔ !(a == b)
  - > ⇔ !(a < b) && (!(a == b))
  - <= ⇔ (a < b) || (a == b)
  - etc.

Assignment Operator

- The 4th (and last) function the compiler provides if you don't
- Syntax: *Type*& operator = (const *Type*&);
- Returns reference for chaining
  - Returns itself (i.e. return *this;)
  - Const reference parameter avoids need for copy constructor
- Should check for assignment to self
  - Avoids problems with pointer members
  - if(this != &RHS) …

Implementing an Assignment Operator

```
class Box {
public:
    Box& operator = (const Box& RHS)
    {
        if( this != &RHS)
        {
            length = RHS.length;
            width = RHS.width;
            height = RHS.height;
        }
        return *this;
    }
};
```

Optimizing Returns

- Creating a temporary object in return statement is excellent optimization
  - return Box(10, 15, 20);
- Not the same as creating a local temp variable and returning it
  - Causes constructor, copy to temp object, destructor calls
- Compiler knows there's no need for a temp object created in a return statement
  - Creates it directly in caller's lvalue location

Unusual Operators

- Index operator []
  - Returns a reference into some internal data
- new and delete
  - Allows custom memory management
  - An advanced topic for a later quarter
- Operator comma
  - Why bother?
- Operator ->
  - Defines “smart pointers” or *iterators*
  - Used frequently in C++ Standard Template Library (summer topic)

More Unusual Operators

- Operator ()
  - Make your object look like a function call
  - Can be overloaded to have many signatures
- Operator ->*
  - "pointer to member"
    - Essentially a pointer to a member function
  - Advanced topic

Outline

- Programming and C++
- C vs. C++
- Input/Output Library
- Object Classes
- Function and Operator Overloading
- <u>Inheritance and Virtual Functions</u>

Inheritance

- Allows extension of one “generic” data type with another more specific version
- The basis for polymorphism
- The heart of Object-Oriented Programming
- Uses the “Is-A” or “Is Kind Of” test

Inheritance Syntax

- Syntax:

class *Derived* : public *Base*
{
…
};

- Example:

```
class Carton : public Box
{
public:
    Carton( const std::string& c_strMaterial );
private:
    const std::string m_strMaterial;
};
```

Member Access Control with Inheritance

- Review:
  - public – Anyone outside the class has access
  - private – No one outside the class has access
  - protected – private to the outside world, but public to derived classes

Order of Construction

- Derived classes are extensions of base classes
  - Base classes must be created before they can be extended
- Base class constructor called by derived class before derived class constructor entered

Order of Destruction

- Derived classes are extensions of base classes
  - Derived classes must be destroyed before base class
- Destructors called from most derived to base

Calling Constructors Explicitly

- So far, base class default constructors called implicitly
- Sometimes you want to call a different base constructor
  - Sometimes base class default constructor doesn't even exist
- Call in member initialization list

Copy Constructors Revisited

- Remember: default constructor called implicitly if nothing else specified
  - Causes base class not to get copied when derived class copy constructor invoked
- Derived class copy constructor must explicitly call base class copy constructor

Base Class Access Specifiers

- Controlled access to members based access specifiers inside base class so far
- Can override member access control for entire class by changing how base class is inherited
- Remember syntax: class *Derived* : public *Base* {…};
- Changing "public" inheritance of base class alters definitions of internal access control

Public Base Class Inheritance

- When inheriting from base classes as “public” the normal definitions of access control pertain
- This is the most common

class *Derived* : public *Base* {…};

Protected Base Class Inheritance

- Inheriting base class as protected limits all public base class members to “protected” in derived class

class *Derived* : protected *Base* {…};

Private Base Class Inheritance

- Inheriting the base class as private makes the entire base class private in the derived class

class *Derived* : private *Base* {…};

Overloading Functions/Members in Inheritance Hierarchies

- What if you reuse a base class's function or member variable name in a derived class?
- Basic scoping rules apply
  - Derived name is "more local"
  - Derived class version hides base class version
  - Use fully qualified name to access base class version

Object Slicing

class Carton : public Box {…};

- A Carton “is a” Box
- Legal to create a box object from a carton:

  Carton thin(“paper”);

  Box paper(thin);

- Paper loses all Carton-related information
  - Only retains data declared in Box
  - Called *object slicing*
- Can only move <u>up</u> a hierarchy tree

Overriding Inherited Behaviors

- What if we want to change behavior in a derived class?
  - e.g. Given a Shape object drawing a square is different than drawing a triangle
  - Make an operation *polymorphic*
  - Done with the *virtual* keyword

Virtual Functions

- Allows the selection of the correct function to be invoked at run-time
- Overload base class function in derived class
  - Same name, parameter list, const
  - Stopping here would only hide the function name, not override it
- Function declared as virtual in base class

  virtual void Draw(void) const;

  - Don’t have to mark as virtual in derived classes but recommended for clarity
- Derived class may or may not implement the function
  - Doesn’t matter—compiler selects most-derived implementation