An Overview of C++

Objectives

- Introduction
- What is object-oriented programming?
- Two versions of C++
- C++ console I/O
- C++ comments
- Classes: A first look
- Some differences between C and C++
- Introducing function overloading
- C++ keywords
- Introducing Classes

Introduction

- C++ is the C programmer's answer to Object-Oriented Programming (OOP).
- C++ is an enhanced version of the C language.
- C++ adds support for OOP without sacrificing any of C's power, elegance, or flexibility.
- C++ was invented in 1979 by Bjarne Stroustrup at Bell Laboratories in Murray Hill, New Jersey, USA.

Introduction (cont.)

- The elements of a computer language do not exist in a void, separate from one another.
- The features of C++ are highly integrated.
- Both object-oriented and non-object-oriented programs can be developed using C++.

What is OOP?

- OOP is a powerful way to approach the task of programming.
- OOP encourages developers to decompose a problem into its constituent parts.
- Each component becomes a self-contained object that contains its own instructions and data that relate to that object.
- So, complexity is reduced and the programmer can manage larger programs.

What is OOP? (cont.)

- All OOP languages, including C++, share three common defining traits:
  - Encapsulation
    - Binds together code and data
  - Polymorphism
    - Allows one interface, multiple methods
  - Inheritance
    - Provides hierarchical classification
    - Permits reuse of common code and data

Two Versions of C++

- A traditional-style C++ program -

```
#include <iostream.h>

int main()
{
        /* program code */
        return 0;
}
```

TWO VERSIONS OF C++ (CONT.)

- A modern-style C++ program that uses the new-style headers and a namespace -

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

int main()
{
        /* program code */
        return 0;
}
```

The New C++ Headers

- The new-style headers do not specify filenames.
- They simply specify standard identifiers that might be mapped to files by the compiler, but they need not be.
  - <iostream>
  - <vector>
  - <string>, not related with <string.h>
  - <cmath>, C++ version of <math.h>
  - <cstring>, C++ version of <string.h>
- Programmer defined header files should end in ".h".

Scope Resolution Operator (::)

- Unary Scope Resolution Operator
  - Used to access a hidden global variable
  - **Example:** usro.cpp
- Binary Scope Resolution Operator
  - Used to associate a member function with its class (will be discussed shortly)
  - Used to access a hidden class member variable (will be discussed shortly)
  - **Example:** bsro.cpp

NAMESPACES

- A namespace is a declarative region.
- It localizes the names of identifiers to avoid name collisions.
- The contents of new-style headers are placed in the **std** namespace.
- A newly created class, function or global variable can put in an existing namespace, a new namespace, or it may not be associated with any namespace
  - In the last case the element will be placed in the global unnamed namespace.
- Example: namespace.cpp

C++ Console I/O (Output)

- cout << "Hello World!";
  - printf("Hello World!");
- cout << iCount; /* int iCount *
  - printf("%d", iCount);
- cout << 100.99;
  - printf("%f", 100.99);
- cout << "\n", or cout << '\n', or endl
  - printf("\n")
- In general, cout << *expression*;

cout ???

Shift right operator ???

How does a shift right operator produce output to the screen?

Do we smell polymorphism here???

C++ Console I/O (Input)

- cin >> strName; /* char strName[16] */
  - scanf("%s", strName);
- cin >> iCount; /* int iCount */
  - scanf("%d", &iCount);
- cin >> fValue; /* float fValue */
  - scanf("%f", &fValue);
- In general, cin >> *variable*;

Hmmm. Again polymorphism.

C++ Console I/O (I/O chaining)

- cout << “Hello” << ‘ ‘ << “World” << ‘!’;
- cout << “Value of iCount is: ” << iCount;
- cout << “Enter day, month, year: ”;
  - cin >> day >> month >> year;
    - cin >> day;
    - cin >> month;
    - cin >> year

What’s actually happening here? Need to learn more.

C++ Console I/O (example)

```
include <iostream>
int main()
{
    char str[16];
    std::cout << "Enter a
    string: ";
    std::cin >> str;
    std::cout << "You entered:
    "                    << str;
}
```

```
include <iostream>
using namespace std;
int main()
{
    char str[16];
    cout << "Enter a string: ";
    cin >> str;
    cout << "You entered: "
                         << str;
}
```

C++ Comments

- Multi-line comments
  - /* one or more lines of comments */
- Single line comments
  - // …

Classes: A First Look

- General syntax -

```
class class-name
{
        // private functions and variables
public:

        // public functions and variables
}object-list (optional);
```

Classes: A First Look (cont.)

- A class declaration is a logical abstraction that defines a new type.
- It determines what an object of that type will look like.
- An object declaration creates a physical entity of that type.
- That is, an object occupies memory space, but a type definition does not.
- **Example:** p-23.cpp, p-26.cpp, stack-test.c.

CLASSES: A FIRST LOOK (CONT.)

- Each object of a class has its own copy of every variable declared within the class (except static variables which will be introduced later), but they all share the same copy of member functions.
  - How do member functions know on which object they have to work on?
    - The answer will be clear when "***this***" pointer is introduced.

Some Differences Between C and C++

- No need to use "void" to denote empty parameter list.
- All functions must be prototyped.
- If a function is declared as returning a value, it ***must*** return a value.
- Return type of all functions must be declared explicitly.
- Local variables can be declared anywhere.
- C++ defines the **bool** datatype, and keywords **true** (any nonzero value) and **false** (zero).

Introducing Function Overloading

- Provides the mechanism by which C++ achieves one type of polymorphism (called **compile-time polymorphism**).
- Two or more functions can share the same name as long as either
  - The type of their arguments differs, or
  - The number of their arguments differs, or
  - Both of the above

Introducing Function Overloading (cont.)

- The compiler will automatically select the correct version.
- The return type alone is not a sufficient difference to allow function overloading.
- **Example:** p-34.cpp, p-36.cpp, p-37.cpp.

Q. Can we confuse the compiler with function overloading?
A. Sure. In several ways. Keep exploring C++.

C++ Keywords (partial list)

- bool
- catch
- delete
- false
- friend
- inline
- namespace
- new
- operator
- private
- protected
- public
- template
- this
- throw
- true
- try
- using
- virtual
- wchar_t

Introducing Classes

CONSTRUCTORS

- Every object we create will require some sort of initialization.
- A class's constructor is automatically called by the compiler each time an object of that class is created.
- A constructor function has the ***same name*** as the class and has ***no return type***.
- There is no explicit way to call the constructor.

DESTRUCTORS

- The complement of a constructor is the destructor.
- This function is automatically called by the compiler when an object is destroyed.
- The name of a destructor is the ***name of its class***, preceded by a ~.
- There is explicit way to call the destructor but highly discouraged.
- **Example** : cons-des-0.cpp

Constructors & Destructors

- For global objects, an object's constructor is called once, when the program first begins execution.
- For local objects, the constructor is called each time the declaration statement is executed.
- Local objects are destroyed when they go out of scope.
- Global objects are destroyed when the program ends.
- **Example**: cons-des-1.cpp

CONSTRUCTORS & DESTRUCTORS

- Constructors and destructors are typically declared as **public**.
- That is why the compiler can call them when an object of a class is declared anywhere in the program.
- If the constructor or destructor function is declared as **private** then no object of that class can be created outside of that class**.** ***What type of error ?***
- **Example**: private-cons.cpp, private-des.cpp

Constructors That Take Parameters

- It is possible to ***pass arguments*** to a constructor function.
- Destructor functions ***cannot*** have parameters.
- A constructor function with no parameter is called the ***default constructor*** and is supplied by the compiler automatically if no constructor defined by the programmer.
- The compiler supplied default constructor ***does not initialize*** the member variables to any default value; so they contain garbage value after creation.
- Constructors ***can be overloaded***, but destructors ***cannot be overloaded.***
- A class can have multiple constructors.
- **Example**: cons-des-3.cpp, cons-des-4.cpp, cons-des-5.cpp, cons-des-6.cpp

Object Pointers

- It is possible to access a member of an object via a pointer to that object.
- Object pointers play a massive role in run-time polymorphism (will be introduced later).
- When a pointer is used, the arrow operator (->) rather than the dot operator is employed.
- Just like pointers to other types, an object pointer, when incremented, will point to the next object of its type.
- **Example**: obj.cpp

IN-LINE FUNCTIONS

- Functions that are not actually called but, rather, are expanded in line, at the point of each call.
- Advantage
  - Have no overhead associated with the function call and return mechanism.
  - Can be executed much faster than normal functions.
  - Safer than parameterized macros. ***Why ?***
- Disadvantage
  - If they are too large and called too often, the program grows larger.

In-line Functions

```
inline int even(int x)
{
    return !(x%2);
}

int main()
{
    if(even(10)) cout << "10 is
    even\n";
    // becomes if(!(10%2))

    if(even(11)) cout << "11 is
    even\n";
    // becomes if(!(11%2))

    return 0;
}
```

- The **inline** specifier is a *request*, not a command, to the compiler.
- Some compilers will not in-line a function if it contains
  - A **static** variable
  - A **loop**, **switch** or **goto**
  - A **return** statement
  - If the function is **recursive**

AUTOMATIC IN-LINING

- Defining a member function inside the class declaration causes the function to automatically become an in-line function.
- In this case, the **inline** keyword is no longer necessary.
  - However, it is not an error to use it in this situation.
- Restrictions
  - Same as normal in-line functions.

AUTOMATIC IN-LINING

```
// Automatic in-lining
class myclass
{
    int a;
public:
    myclass(int n) { a = n; }
    void set_a(int n) { a = n; } int
    get_a() { return a; }
};
```

```
// Manual in-lining
class myclass
{
    int a;
public:
    myclass(int n);
    void set_a(int n);
    int get_a();
};
inline void myclass::set_a(int n)
{
    a = n;
}
```

Lecture Contents

- Teach Yourself C++
  - Chapter 1 (Full, with exercises)
  - Chapter 2.1, 2,2, 2.4, 2.6, 2.7