Introduction to C++ (Extensions to C)

C is purely procedural, with no objects, classes or inheritance. C++ is a hybrid of C with OOP!

The most significant extensions to C are:

- much **stronger type checking**. Missing casts that produce warnings in C produce errors in C++
- the introduction of **true O-O classes**
- a **formal inheritance mechanism** in which derived classes can specialize parent classes
- formal support for **polymorphic behavior** in which a derived class may override a base class method simply by providing a method of the same name.
Extensions to C (cont'd)

- support for **function overloading** in which there may be different implementations with a single function name.
- an **operator overloading** mechanism that is analogous to function overloading
- the ability to pass parameters **by reference** in addition to the standard pass by value.
- yet another **input/output library** that may be used in addition to standard and low level I/O
The Parts of a C++ Program

// sample C++ program
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, there!";
    return 0;
}

Note: C++ files have a .cpp extension and are compiled using g++
Interactive I/O using cin and cout

- C++ uses streams to perform input/output operations
- A stream is an object that allows a program to insert characters to it or extract characters from it.
- The standard input/output stream objects are defined in the header file `iostream`.
- `<iostream>` must be included for any program that outputs data to the screen or inputs data from the keyboard using C++-style stream input/output.
Interactive I/O using cin and cout

With `<iostream>` you automatically get the following objects:

- **cout** – a predefined object of the ostream class that displays data to standard output; corresponds to stdout. Used

- **cin** – a predefined object of the istream class that reads data from standard input; corresponds to stdin

- **cerr** – a predefined object of the ostream class that represents the standard error stream; corresponds to stderr.

- **clog** – a predefined object of the ostream class represents the standard logging stream; is associated with stderr.

- **endl** – outputs an end of line character and flushes the buffer.
The `cout` Object

- Displays information on the computer screen
- Use `<<`, the insertion operator, to send information to `cout`
  ```cpp
  cout << "Hello, there!";
  ```
- Can use `<<` to send multiple items to `cout`
  ```cpp
  cout << "Hello, " << "there!";
  ```
  Or
  ```cpp
  cout << "Hello, ";
  cout << "there!";
  ```
The `cout` object

Example

```cpp
cout << "Hello";
cout << " C++ is great!";
cout << endl;
```

Or

```cpp
cout << "Hello" << " C++ is great!" << endl;
```
Starting a New Line

• To get multiple lines of output on screen
  - Use `endl`
    
    ```cpp
    cout << "Hello, there!" << endl;
    ```
  - Use `\n` in an output string
    
    ```cpp
    cout << "Hello, there!\n";
    ```
Formatting Output

• Can control how output displays for numeric and string data
  – size
  – position
  – number of digits
• Requires `iomanip` header file
Stream Manipulators

- Used to control features of an output field
- Some affect just the next value displayed
  - `setw(x)` : Print in a field at least $x$ spaces wide. It will use more spaces if specified field width is not big enough.
Stream Manipulators

- Some affect values until changed again
  - **fixed**: Use decimal notation (not E-notation) for floating-point values.
  - **setprecision**(x):
    - When used with **fixed**, print floating-point value using x digits after the decimal.
    - Without **fixed**, print floating-point value using x significant digits.
  - **showpoint**: Always print decimal for floating-point values.
  - **left, right**: left-, right justification of value
Manipulator Examples

```cpp
const float e = 2.718;
float price = 18.0;
cout << setw(8) << e << endl;
cout << left << setw(8) << e << endl;
cout << setprecision(2); 
cout << e << endl;
cout << fixed << e << endl;
cout << setw(6) << price;
```

Displays

```
2.718
2.7
2.72
18.00
```
The `cin` Object

- Standard input object
- Like `cout`, requires `iostream` file
- Used to read input from the keyboard
- Often used with `cout` to display a user prompt first
- Data is retrieved from `cin` with `>>`
- Input data is stored in one or more variables
The `cin` Object

- Line at a time input
- User input goes from keyboard to the input buffer, where it is stored as characters
- `cin` converts the data to the type that matches the variable

```cpp
int height;
cout << "How tall is the room? ";
cin >> height;
```
The `cin` Object

- Can be used to input multiple values

\[
\text{cin} \gg \text{height} \gg \text{width};
\]

Or

\[
\text{cin} \gg \text{height} \\
\quad \gg \text{width};
\]

- Multiple values from keyboard must be separated by spaces or [Enter]
- Must press [Enter] after typing last value
- Multiple values need not all be of the same type
- Order is important; first value entered is stored in first variable, etc.
Namespaces in C++

• In C++, as in other languages, variables, functions, and objects are program entities that must have names.
• C++ uses namespaces to organize the names of program entities.
• Elements within a namespace may be accessed using the scope resolution operator (::). For example, \textit{std::cout} refers to the cout stream that is a member of the \textit{std} namespace.
Namespaces in C++

• To avoid repeatedly having to type namespace qualifiers, C++ provides a directive called `using`, which creates an alias to some element of another namespace. For example:

```cpp
int sample() {
using std::cout;
cout << "This is a test. " << std::endl;
}
```
Namespaces in C++

• For frequently-used namespaces, such as \textit{std}, a special directive can be placed at the top of the file:

\begin{verbatim}
using namespace std;
\end{verbatim}

• This directive makes every element inside the \textit{std} namespace available in the current (usually the global) namespace.

\begin{verbatim}
using namespace std;
int main()
{
    cout<< "This is a test. " << endl;
}
\end{verbatim}
Type Casting

- Used for manual data type conversion
- Format
  
  ```
  static_cast<Data Type>(Value)
  ```
- Example:
  
  ```
  cout << static_cast<int>(4.2);
  // Displays 4
  ```
char ch = 'C';
cout << ch << " is stored as " << static_cast<int>(ch);

gallons = static_cast<int>(area/500);

avg = static_cast<double>(sum)/count;
double Volume = 21.58;
int intVol1, intVol2;
intVol1 = (int) Volume; // C-style
    // cast
intVol2 = int (Volume); // Prestandard
    // C++ style
    // cast

C-style cast uses **prefix notation**
Prestandard C++ cast uses **functional notation**
**static_cast** is the current standard
The `bool` Data Type

- Represents values that are **true** or **false**
- `bool` values are stored as integers
- **false** is represented by 0, **true** by 1

```plaintext
bool allDone = true;
bool finished = false;
```

```
+---+---+
| 1 | 0 |
+---+---+
```