HelloWorld.cpp

#include <iostream>  // C++ I/O library
using namespace std;  // To use ostream object called cout
int main(){
    // entry point of any C++ program
    // Prints "HelloWorld" to screen
    cout << "HelloWorld";
    return 0;
}

Overview

- Compiling C++ programs
- Data Types, Pointers, and References
- Classes
  - Interface and Implementation
  - Constructors, Destructors
  - Member functions
  - Const
  - Enumerations
  - Operator overloading
  - Namespaces
  - I/O

Compiling C++ Programs

- Compile using g++ which is the GNU C++ compiler
- Source file end in .cpp
  - Some platforms use .cc or .C
  - .C can be confused with .c (C source) if platforms are not case sensitive about source my names
- Simple Compile: g++ filename.cpp
  - Generates ./a.out
  - Has all options that we have seen before in C
    - -g, -Wall, -c, -o, -s
- One of the source files must contain the function main
  - int main() or main(int argc, int * argv[])
Header Files

- Contains class definitions and function prototypes
- In source the header is included using #include
- Usually header files use include guards to prevent a header from being included more than once in the same compilation unit
- Example in hello.h

```c
#ifndef HELLO_H
#define HELLO_H

void hello();
void goodbye(int);
#endif
```

Makefile

```makefile
CPP = g++
FLAGS = -g
EXEC = hello
OBJS = hello.o main.o
$(EXEC): $(OBJS)
	$(CPP) $(FLAGS) -o $(EXEC) $(OBJS)
hello.o: hello.cpp hello.h
	$(CPP) $(FLAGS) -c hello.cpp
main.o: main.cpp hello.h
	$(CPP) $(FLAGS) -c main.cpp
clean:	rm -f $(EXEC) $(OBJS)
```

Data Types

- Similar to C except the bool for Boolean

- bool is a single bit:
  - 1 for true
  - 0 for false

- Can be used as an integer
  - bool check = true;
  - int add = 2 + check;
  - check = check - 1;

References

- A reference is an alias

- A reference is different from a pointer
  - A reference cannot be null
  - Once established, a reference cannot be changed
  - It does not need to be de-referenced
  - All operators operate on referenced values

- A reference could be thought of as a permanently de-referenced pointer
References contd ..

- A reference is declared using the ampersand
  ```
  int i = 8;
  int & j = i;  // j is now an alias for i
  j++;  // i is now 9
  i += 5;  // i and j are 14 now
  j = j * 2;  // i and j are 28
  ```

- Most common use is in functions - allow the function to change the actual value a.k.a Pass by Reference
  ```
  void swapnum(int &i, int &j) {
    int temp = i;
    i = j;
    j = temp;
  }
  ```

References contd ..

- Consider the following Java class
  ```
  public class MyObject {
    public int val;
  }
  ```

- Consider the following statements
  ```
  MyObject obj1 = new MyObject();
  MyObject obj2;
  obj1.val = 12;
  obj2 = obj1;
  obj1.val = 18;
  obj1.val == obj2.val // true or false?
  ```

References contd..

- Consider the following C++ version of MyObject
  ```
  class MyObject {
    public:
      int val;
  }
  ```

- Consider the following statements
  ```
  MyObject obj1;
  MyObject obj2;
  obj1.val = 12;
  obj2 = obj1;
  obj1.val = 18;
  obj1.val == obj2.val // true or false?
  ```

References contd..

- Java uses reference semantics for assignments, so when the code is run, both obj1 and obj2 are variables that end up referring to the same object

- C++ uses copy semantics for assignments, so when the code is run obj1 and obj2 are two different objects with different values
  ```
  MyObject obj1;
  obj1.val = 12;
  MyObject obj2 = obj1;
  obj1.val = 18;
  ```
Class

- A class directly represents a concept in a program
  - If you can think of "it" as a separate entity, it is plausible that it could be a class or an object of a class
  - E.g. vector, input stream, string, graph, window etc.

- A class is a (user-defined) type that specifies how objects of its type can be created and used

- In C++ (as in most modern languages), a class is the key building block for large programs

Class: Member Access

```cpp
class Date {
    int y, m, d; // year, month, day

    Date(int y, int m, int d); // constructor: check for valid date and initialize

    // access functions:
    void add_day(int n); // increase the Date by n days
    int month();
    int day();
    int year();
};

Date my_birthday(1950, 12, 30);
cout << my_birthday.month() << endl; // we can read
my_birthday.m = 14; //error: m is private by default
```

Class

- A class is a user-defined type
  ```cpp
class X { // this class' name is X
    private: // private members -- that's the implementation details
        // functions
        // types
        // data
    public: // public members -- that's the interface to users
        // functions
        // types
        // data (often best kept private)
  };
  ```

Class: Member access

- Why bother with the public/private distinction?
- Why not make everything public?
  - To provide a clean interface
    - Data and messy functions can be made private
  - To maintain an invariant
    - Only a fixed set of functions can access the data
  - To ease debugging
    - Only a fixed set of functions can access the data
    - (known as the "round up the usual suspects" technique)
  - To allow a change of representation
    - You need only to change a fixed set of functions
    - You don't really know who is using a public member
Class: Implementation

// In Date.h contains the class interface
class Date {
public:
  Date(int y, int m, int d);
  void add_day(int n);
  int month();
  // ...
private:
  int y, m, d;
};
// In Date.cpp contains the class implementation
Date::Date(int yy, int mm, int dd):y(yy), m(mm), d(dd){ /* ... */};
Date::month() { return m; }

■ Implementation of constructor or member functions
  ■ ClassName :: Method to indicate "member of"
  ■ :y(yy), m(mm) is known as member initializer

Class: Constructor

Date::Date(int yy, int mm, int dd)
  : y(yy), m(mm), d(dd) // initialize data members
  {
    if (!check(y, m, d)) throw Invalid(); // check for validity
  }

■ A default constructor is used if you do not declare and implement one
  ■ Constructor with no arguments
  ■ Usage: Date d;

■ Note: It is not allowed to call one constructor from another constructor

Class: Common Errors

int month() { return m; }
  ■ error: forgot Date ::
  ■ this month() will be seen as a global function not the member function, can’t access members

int Date::season() /* ... */
  ■ error: no member called season

Enumerations

■ An enum (enumeration) is a very simple user-defined type, specifying its set of values (its enumerators)

■ For example:
  enum Month {
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
  };
  Month m = feb;
  m = 7;  // error: can’t assign int to Month
  int n = m;  // ok: we can get the numeric value of a Month
  Month mm = Month(7);  // convert int to Month (unchecked)
Enumerations

- Simple list of constants:
  ```
  enum { red, green }, // the enum { } doesn't define a scope
  int a = red, // red is available here
  enum { red, blue, purple }, // error: red defined twice
  ```

- Type with list of constants
  ```
  enum Color { red, green, blue, /* ... */ };
  enum Month { jan, feb, mar, /* ... */ };
  Month m1 = jan;
  Month m2 = red; // error, red isn't a Month
  Month m3 = 7; // error, 7 isn't a Month
  int i = m1; // ok: an enumerator is converted to its value, i==0
  ```

Enumerations – Values

- By default
  - The first enumerator has the value 0.
  - The next enumerator has the value "one plus the value of the numerator before it"
    ```
    enum { horse, pig, chicken }, // horse==0, pig==1, chicken==2
    ```

- You can control numbering
  ```
  enum { jan=1, feb, mar /* ... */ }; // feb==2, mar==3
  ```

- You can control numbering
  ```
  enum stream_state { good=1, fail=2, bad=4, eof=8 };
  int flags = fail+eof; // flags==10
  stream_state s = flags; // error: can't assign an int to a stream_state
  stream_state s2 = stream_state(flags); // explicit conversion (be careful!)
  ```

 Enums with Classes

- Simple Date (use Month type)
  ```
  class Date {
       public:
          enum Month {
             jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
          };
          Date(int y, Month m, int d); // constructor
          // ...
       private:
          int y; // year
          Month m;
          int d; // day
       };
  
  Date my_birthday(1950, Date::dec, 30);
  ```

Const: Variables that don’t change

- A variable
  ```
  Date d(2004, Date::jan, 7); // a variable
  ```

- A constant
  ```
  const Date d2(2004, Date::feb, 28); // a constant
  ```

- A variable
  ```
  d2 = d; // error: d2 is const
  d2.add(1); // error: d2 is const
  ```

- A constant
  ```
  d = d2; // fine
  d.add(1); // fine
  ```

- Should work if and only if f() doesn’t modify d2
  ```
  d2.f(); // should work if and only if f() doesn’t modify d2
  ```

- How do we achieve that? (Say that’s what we want, of course)
**Const : member functions**

- Specifies that the function is a "read-only" function that does not modify the object for which it is called

```cpp
class Date {
public:
    // ...
    int day() const { return d; } // const member: can't modify
    void add_day(int n); // non-const member: can modify
    // ...
};
```

```cpp
Date d1(2000, Date::jan, 20);
const Date d2(2001, Date::feb, 21);
cout << d1.day() << " – " << d2.day() << endl; // ok
```
**Why pass a const reference to a function?**

```cpp
date next_sunday(const date& d) { .. }
```

- Reference allows you to change variable from method
- But const will disallow that
- So why do this?
  - To avoid copy of data fields on the function stack

**Operator overloading**

- You can define almost all C++ operators for a class or enumeration operands
  - That’s often called “operator overloading”

```cpp
bool operator==(const date& a, const date& b)
{
    return (a.year() == b.year() 
    && a.month() == b.month() 
    && a.day() == b.day());
}
```

- You can define only existing operators
  - E.g., + - * / % [] () ^ ! & < <= > >=
- You can define operators only with their conventional number of operands
- An overloaded operator must have at least one user-defined type as operand
  - `int operator+(int, int);` // error: you can’t overload built-in +
  - `Vector operator+(const Vector&, const Vector &);` // ok
- Advice (not language rule):
  - Overload operators only with their conventional meaning
  - + should be addition, * be multiplication, [] be access, () be call, etc.
- Advice (not language rule):
  - Don’t overload unless you really have to

**Namespaces**

- Restrict visibility of a name
  ```cpp
  namespace foo{
  int x;
  void bar() { .. }
  }
  ```
- Access members with scope resolution operator `foo::bar()` or `foo::x`
  - Global namespace: nothing on lhs of `::`
    - `::somevariable`
- The using keyword opens a namespace:
  - `using namespace foo;`
**IO C++ Style**
- printf/fgets still work, but...
- C++ uses iostreams, which overload << and >>

```
#include <iostream>  // declares std::cout, std::endl
using namespace std; // can skip std::
int main() {
    int x;
    cout << "Enter an integer" << endl; // endl inserts a newline char
    cin >> x; // object of class istream that represents the std input stream
    return 0;
}
```

**I/O: Output Streams (ostream)**
- Object of type ostream converts values into sequences of characters which are then output somewhere
- << operator is used to send values to an ostream
- iostream class predefines ostream object called `cout` that sends text to stdout
- Also has member functions. Examples:
  - cout.put('c')
  - cout.flush()

**I/O: Input Stream (istream)**
- istream object reads sequences of characters from somewhere and converts them to values
- >> operator is usually used to read values from an istream
  - By default it skips over white space before reading a value
  - eof() function is used to detect the end of the input file
- iostream class predefines istream object called `cin` reads from stdin
- Other member function
  - cin.read(buffer, size)
  - cin.get()
  - cin.get(buffer, size) // overloaded function

**Essential Operations**
- Default constructor (defaults to: nothing)
  - No default if any other constructor is declared
- Copy constructor (defaults to: copy the member)
  - To make copies (clone)
- Copy assignment (defaults to: copy the members)
  - Assignment operator
- Destructor (defaults to: nothing)
- For example
  ```
  Date d; // default constructor
  Date d2 = d; // ok: copy initialized (copy the elements)
  d = d2; // ok copy assignment (copy the elements)
  ```
Class: Copy Constructor

- To make a copy of an object value
- Takes an instance of the same class as a constant reference argument

```cpp
Date(const Date & d) {
    y = d.y;
    // ...
}
```

- A copy constructor is often called implicitly such as when passing by value

```cpp
Date d(...);
doStuff(d);  // Copy constructor called
```

new/delete operator

- Use new operator for dynamic allocation
  - It returns a pointer to the beginning of the new block of memory allocated
  - An exception of type `bad_alloc` is thrown when the allocation fails. If it is not handled by a specific handler, the program execution is terminated

```cpp
Date * d = new Date(1990, 12, 3);
```

- Use `delete` for de-allocating dynamic data
  - `delete d;`
  - To delete array: `delete [] arrayname`

- malloc-new and free-delete have several similarities:
  - both `malloc()` and `new` return a pointer that is suitably aligned for a given machine architecture
  - both `free()` and `delete` do nothing with a NULL pointer

Class: Destructor

- Define using tilde following class name and takes no arguments:

```
~Date()
```

- It is automatically called when an object is destroyed, either because its scope of existence has finished
  - If it was defined as a local object within a function and the function ends
  - If it is an object dynamically assigned and it is released using the operator `delete`

Destructor Example

```cpp
class MyStorage{
    private:
        int * space;
    public:
        MyStorage(int s){ space = new int[s]; }
        int & operator[] (int i){ return space[i]; }
        ~MyStorage(){delete [] space;}
};
```
**Inlining**
- When a method or constructor/destructor body is implemented inside the class definition
- Allows faster code execution as it avoids the overhead of a function call
- Can make the compiled code larger and more complex
- Use for very short work
  - Never use them with loops or recursive calls

**Forward Declaration**
- A class must be defined before it is used
  - If a class name is used in another class definition, the class must be defined prior to the use
  - This can be a problem if the first class also uses the name of the second class
- A forward declaration is used to declare the name of class
  - Permits pointers to the class to be declared
  - Cannot invoke methods in the class (since they are not defined yet)

**Keyword this**
- Every method has a pointer name `this` which points to the object the methods was invoked on
- Example
  ```cpp
  Date & doStuff() {
    this->m = 1;
    return *this;
  }
  ```

**File I/O**
- `fstream` provides an interface to read and write data from files as input/output streams
- Member functions
  - `open`
    - Can open files in many modes e.g. binary, append, in, out etc.
    - To use these modes do: `fstream::binary`
  - `close`
  - `is_open`
fstream Example

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

int main(){
    fstream f1, f2;
    char c;
    f1.open("input.txt", fstream :: in);
    f2.open("output.txt", fstream :: out | fstream :: app);
    if(!f1.is_open() || !f2.is_open()){
        cout << "Error opening file" << endl;
        return -1; //error
    }
    while (!f1.eof()){
        f1.get(c);  //read a character from stream f1
        cout << c;  //write character to stdout
        f2 << c;    //write the character to stream f2
    }
    f1.close();
    f2.close();
    return 0;
}
```

Formatted I/O

- `iostream` and `fstream` inherit member functions from `io_base` class
- Most commonly used:
  - `precision(int n)`
  - `width(int n)`
  - `setf(ios :: flag)`

width

- `width` set the minimum width of the column
- If more space is needed to print a value, the entire value is printed
- Calling `width` affects only the next item that is sent to the output stream
- Example
  ```cpp
  int count = 5;
  cout.width(10)
  cout << "Count = " << count << endl;
  ```
setf

- Short for set flags
- Some useful flags
  - ios :: right
    - When used with width forces the value to be right justified
    - Automatically unsets left
    - Set by default
  - ios :: left
  - ios :: showpos
    - Forces the + sign to be printed with positive values

Manipulators

- Another way of I/O formatting
  - #include <iomanip>
- Manipulators are global functions designed to be used together with insertion (<<) and extraction (>>) operators performed on iostream stream objects
- Usage
  ```cpp
cout << setw(10) << setbase(16) << val << endl;
```
- setw(int n) works same as width(int n) from io_base class

Nested Class

- One class can be defined within another class
  - If nested class is in defined in the private section, then only the outer class knows about it
  - To access a nested class from outside the outer class (if not in private section), a fully qualified name must be used
    - Outclassname :: InnerClassName variablename

UML

- Unified Modeling Language
- Used to specify, visualize, document artifacts of an object-oriented software under development
- Example class diagram
  - Models classes in a system and how they are related
  - Visibility Modifiers:
    - Public (+)
    - Private (-)
    - Protected (#)
UML Class Diagram Example

Date
- day : int
- year : int
+ day() : int
+ add_day() : void