CIS 330
C/C++ and Unix

Lecture 8

Objects

Object Oriented Programming – Overview

In C++

- Everything is an object
- A program = a bunch of objects telling each other what to do by sending messages
- Each object is made up of other objects (or fundamental data types)
- Every object has a (data) type (also known as **Class**) and those with the same types can receive the same messages

Object has an Interface

- Class describes a set of objects that have the same characteristics (i.e., data elements) and behavior (i.e., functionality).
- You can communicate with an object through its interface (i.e., send a message)
  - Make requests (e.g., tell it what to do)
  - Change its state

Example Class

```
1. class Human {
2. public:
3.     void eat();
4.     void sleep();
5.     void drink();
6.     int getAge() {return age; }
7. private:
8.     int hp;
9.     int mp;
10.    int str;
11.    int age;
12. };
13. Human John;  /* an object */
```

Interface

Class Implementation

- Only expose what would be necessary for programmers (who are using the class) via **encapsulation**
  - Reduces problems (if classes are used in a way that it was not intended to be used)
  - Making changes to the class will not impact its users (assuming the changes are done correctly)
- Each class has three "boundaries"
  - Public – the interface that every object can use
  - Private – no one can access these directly (unless you are the creator); causes compile time error if you try
  - Protected – similar to private but accessible by an inheriting class (more on inheritance later)

Example Class

```
1. class Human {
2. public:
3.     void eat();
4.     void sleep();
5.     void drink();
6.     int getAge() { return age; }
7. private:
8.     int hp;
9.     int mp;
10.    int str;
11.    int age;
12. };
13. Human John;
```

Private

Reusing the Implementation

- A Class is a "unit" of code
- Instantiation of a class = object
  - You can create multiple instances of a class by creating multiple objects of the same class
  - *Class = data type* & *Object = variable*
- You can also relate classes to create new ones
  - Composition
  - Inheritance

Composition

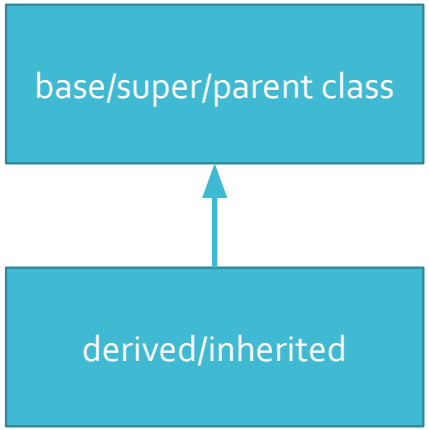
```
1.  class Birthdate {
2.      public:
3.          int getAge(int cur_day, int cur_month,
4.                     int cur_year);
5.      private:
6.          int day;
7.          int month;
8.          int year;
9.  };
```

Composition

Has-a relationship (e.g., a car has an engine, if there is an engine class and a car class)

```
1.  class Human {
2.  public:
3.      void eat();
4.      void drink();
5.      void sleep();
6.      int getAge(int d, int m, int y) {
7.                  return bday.getAge(d, m, y);
8.      }
9.  protected:
10.     int hp;
11.     int mp;
12.     int str;
13.     // int age;
14.     Birthdate bday;
15. };
```

Inheritance



- Any changes to the base class becomes reflected in the derived class
- You can have multiple derived classes from one base type

Inheritance

- Derived class is the same type as the base class (**type equivalence**)
- They have the same interface
- You can add new functions to the interface
- Or change the behavior of existing ones (override)

Example Class

```
1. class Human {
2. public:
3.     void eat();
4.     void sleep();
5.     void drink();
6.     int getAge() { return age; }
7. private:
8.     int hp;
9.     int mp;
10.    int str;
11.    int age;
12. };
```

Inheritance

```
1.  class King : public Human {
2.      public:
3.          bool hasQueen() { return married; }
4.      private:
5.          bool married;
6.  };
7.  King John;
```

Is-a vs. Is-like-a Relationship

Is-a

- Inheritance overrides ONLY base-class functions (i.e., not add any new ones)
- Derived class is EXACTLY like the base class
- e.g., a circle "is-a" shape
  - They both have area as a property

Is-like-a

- Add new implementations to a derived type (in addition to existing ones)
- The new type "is-like-a" base type
- e.g., Heat pump "is-like-a" air conditioner
  - AC cools, but heat pump can both cool and heat (so heat pump class can inherit AC class, then add heating as a functionality)

Polymorphism

- Allows different code to be executed for a given member function depending on the *type of the object*
- Early binding
  - The linker resolves a call to a function to find the absolute address of the *appropriate* code to be executed (C)
- Late binding
  - The function to be executed is determined at runtime – the compiler only ensures that the function exists and type checks the function return value and its parameters (C++)

Polymorphism

```
1.  class Human {
2.  public:
3.      ...
4.      virtual void getStr()
5.      {
6.          cout << "Human has strength " << str << endl;
7.      }
8.      ...
9.  };
```

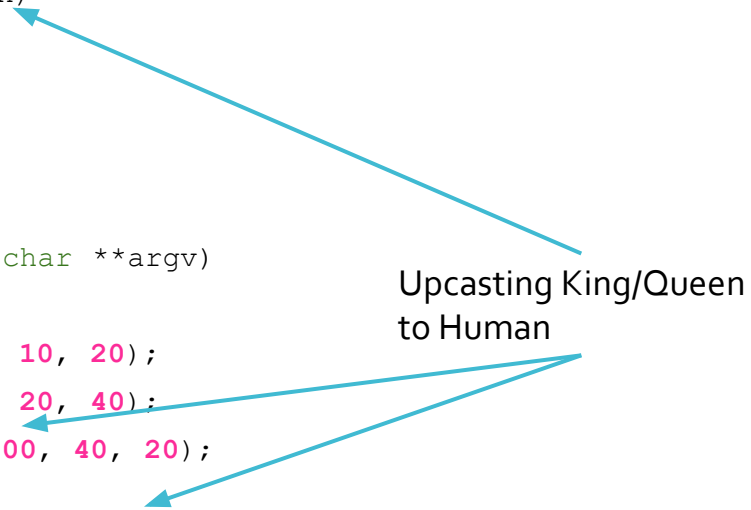
Polymorphism

```
1.  class King : public Human {
2.  public:
3.      ...
4.      void getStr()
5.      {
6.          cout << ”King has strength ” << str << endl;
7.      }
8.      ...
9.  };
10. class Queen : public Human {
11. public:
12.     ...
13.     void getStr()
14.     {
15.         cout << ”Queen has strength ” << str << endl;
16.     }
17.     ...
18. };
```

Polymorphism & Upcasting

- Derived class is the same type as the base class (**type equivalence**)

```
1.   void getStr(Human& h)
2.   {
3.       h.getStr();
4.   }
5.
6.   int main(int argc, char **argv)
7.   {
8.       Human John(100, 10, 20);
9.       King Zelda(200, 20, 40);
10.      Queen Zeldina(200, 40, 20);
11.      getStr(Zelda);
12.      getStr(Zeldina);
13.  }
```



Class

- You can also define the function at class definition
  - The compiler will consider this an inline function – whenever it is called, it simply replaces the call with the source code for that function
- Otherwise, it is called like a regular function (when it is defined outside)

Polymorphism

```
1.  class King : public Human {
2.  public:
3.      ...
4.      void getStr()
5.      {
6.          cout << "King has strength " << str << endl;
7.      }
8.      ...
9.  };
10. class Queen : public Human {
11. public:
12.     ...
13.     void getStr()
14.     {
15.         cout << "Queen has strength " << str << endl;
16.     }
17.     ...
18. };
```

Pointers to Class

- Works similarly to pointers to struct
- You can initialize it when you dynamically allocate it (using something called a constructor, more on it later)

Class as struct

- Classes can be defined using struct (or union)
  - Remember – struct in C++ can have functions
- Difference
  - Members are **public** by default when defined using struct (or union)
  - Members are **private** by default when defined using class

Strings

- Take care the low level manipulation of character arrays
- `#include <string>`
- `string str1;`
- `string str2;`
- `str1 = "Hello"; // no more malloc`
- `str2 = "World"; // or strcpy`
- `str1 = str1 + " " + str2;`
- `cout << str1 << endl;`

Character Sequences

Array of characters are not the same as strings in the standard library

They can be interchangeably used in most cases

Differences

- Character arrays have fixed size (either by array declaration or by assigning it a literal at initialization)
- Library strings have dynamic sizes (more on this later) whose size is determined at runtime (vs. compile time for char arrays)

Conversion

```
1.   char mycstr[] = "some text";
2.   string mystring = mycstr;
3.   cout << mystring;
4.   cout << mystring.c_str();
```

Strings

size	Return length of string (public member function)
length	Return length of string (public member function)
max_size	Return maximum size of string (public member function)
resize	Resize string (public member function)
capacity	Return size of allocated storage (public member function)
reserve	Request a change in capacity (public member function)
clear	Clear string (public member function)
empty	Test if string is empty (public member function)
shrink_to_fit	Shrink to fit (public member function)

Strings

c_str	Get C string equivalent (public member function)
data	Get string data (public member function)
copy	Copy sequence of characters from string (public member function)
find	Find content in string (public member function)
rfind	Find last occurrence of content in string (public member function)
find_first_of	Find character in string (public member function)
find_last_of	Find character in string from the end (public member function)
find_first_not_of	Find absence of character in string (public member function)
find_last_not_of	Find non-matching character in string from the end (public member function)
substr	Generate substring (public member function)
compare	Compare strings (public member function)

Strings

```
string myStr = "hello";
for(int i = 0; i < myStr.size(); i++) {
    cout << myStr[i] << " ";
}
cout << endl;

for(string::iterator i = myStr.begin(); i !=  myStr.end(); i++) {
    cout <<  *i << " ";
}
cout << endl;
```

Vectors

- Strings for numbers – good for dynamically sized arrays
- Vectors class is a template – it can be efficiently applied to different data types

Vector Example

```
1.  vector<string> vec;
2.  ifstream fin("lecture09b.cc");
3.  string line;
4.  while(getline(fin, line)) {
5.      vec.push_back(line);
6.  }
7.  for(int i = 0; i < vec.size(); i++) {
8.      cout << i << ": " << vec[i] << endl;
9.  }
```

Dynamic Memory

- Dynamic memory is allocated and changed as needed during program execution (on the heap)
- Dynamic memory is allocated using the operator ***new***

```
1.  int *intPtr = new (nothrow) int [128];
2.  if(intPtr == nullptr) {
3.      cerr << "Error allocating memory." << endl;
4.  }
```

or

```
1.  int *intPtr = new int [128];
```

- When memory allocation fails, an exception is thrown. A function/code for handling this exception is required.

Dynamic Memory

- When the memory is no longer needed – delete it
  - `delete [] intPtr;`

- How is it different from using malloc?
  - They essentially perform the same functionality
  - Some differences…
    - Exceptions (C can also have exceptions but it is not built into C)
    - new is an **operator** and malloc is a **function** (more on operator overloading…)

- Mixing malloc and new is **NOT** recommended

Data Structures

- `struct` keyword is **NOT** necessary to **create a variable of struct** type
- Can contain functions (which is not allowed in C)
- Can have static members (which is not allowed in C)
  - Remember that static variables can retain its value even when outside of its scope – i.e., once declared and initialized inside a function, it remembers its value even when the function finishes
- Other object oriented properties.

Other Data Types

- `typedef` – still works
- You can also use `using`
  - `using ftype = double;`
- Unions and enumerated types work the same as in C
  - union, enum keywords not required (as in the case of struct) to declare variables
- Anonymous unions
  - Unions with no name
  - Can be accessed directly without the name.
- Enumerated types with enum ***class*** (instead of data type)
  - Does not translate to integers (regular enum has implicit integer value associated to it, or can be specifically assigned)

File IO

```
1. ifstream fin("lecture09b.cc");
2. ofstream fout("lecture09b.tmp");
3. string s;
4. while(getline(fin, s)) {
5.     fout << s << endl;
6. }
```

Live Coding

Test Inheritance and Polymorphism

Live Coding

Test dynamic memory, struct, and file IO