Generalized Class Design

with examples in c++

Author: Owen Stranathan

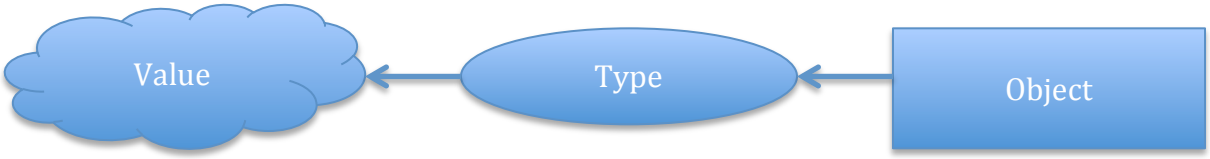
Purpose

The purpose of this paper is to give a clear foundation for new programmers, to venture into the realm of Object Oriented Programming, and specifically focus on the concept of class design.

Introduction

To begin, I will provide you with some definitions, and derive further meaning from them. **Programming** is the act of clearly defining and understanding a problem then formulating and implementing a solution via a computer. **Object oriented programming** is programming, with emphasis on the use of programmer-defined types and objects to solve problems. This second definition is of particular interest, because it uses the words "type" and "object". What is a type and what is an object? A type, insofar as computer programming is concerned, is a name or definition given to a representation of something that exists or has some value, in the "real" world. So then, an object is the aforementioned representation of a "real" world value.

EX1

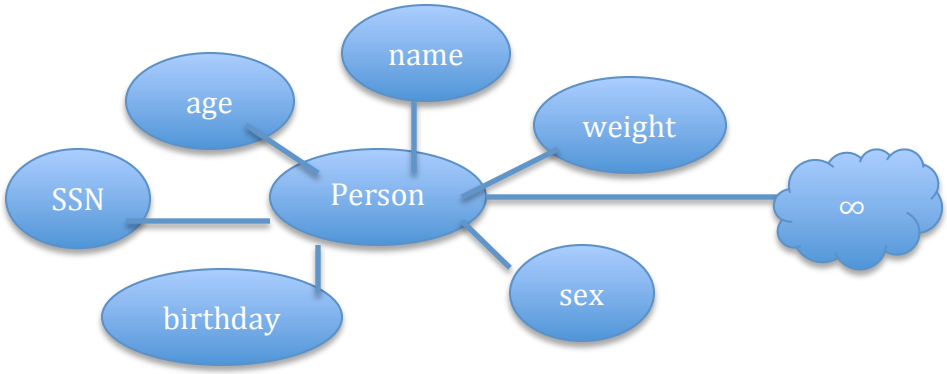


So the type references the value and the object references the type. Put simply, an object is defined by it's type and a type is defined by the value it represents. An example is an object of type person. Being people we have keen insight on the value of people. A person is distinguished by specific values (name, age, weight, etc.) without these values no one person is distinct. So the type of "Person" gives reference to the values a person represents (name, age, weight, etc.) and the object person is a single embodiment (instance), of those associated values. Understanding these concepts of value, type and object is fundamental to understanding object oriented programming.

Abstraction

Programming in c++, you will spend a considerable amount of time designing **classes**. In programming we use classes as a way of defining a **type**, which is turn defines an object. The key to defining a class well is creating an **abstraction** of the type we wish to represent with our class. An abstraction in this context means a definition of what **value** our **object** represents, what our **object** does, and how our **object** interacts with its surroundings (other objects) in the most general terms. For example if we wanted to make an abstraction of the object person we could create something like this:

EX2



This concept map shows the, albeit limited, abstraction of a person. You can create your own abstraction any way you like; I prefer lists with indentations, or drawings. But you can use anything so long as you make the information clear and understandable. Notice the dreamy cloud with the infinity symbol in it? An object in reality can represent an infinite set of values. So when we create an abstraction we must be sure to only consider information relevant to our purpose. The "purpose" is determined by the application for which you wish to use your class.

EX3

Suppose you want to make a person class for a school database, in such an application you would be less interested in including things like weight and birthday, in your abstraction, and more interested in name, age, sex, and possibly SSN (although Student ID would be more likely). Including something like weight would be of limited use, and in the context of a school database, would have very little interaction. So it is important to consider the application when creating an abstraction.

To recap:

We have defined **types** as abstract representations of a real world value, and we know that **objects** are instances of types, (in the context of programming – the representation of the type in memory). We have defined the term **abstraction** and have given sample abstractions for a Person type (class) The next section will discuss the concept of Encapsulation and why it is important to class design.

Encapsulation

Before we begin with encapsulation, Let's just think about what a class in c++ actually does for us. Consider for a moment the following c++ declaration:

```
3
4   int number;
5
```

What does this mean to the computer?

The answer is "Allocate 4 bytes in memory for an object of type integer ". There's that word type again. Knowing what we know about types, we can say that an integer type gives reference to a set of real world values right? Of course we can, the "real" world set of integers (denoted in mathematics as $\mathbb{Z}$) is the set including 0, the Natural numbers (1, 2, 3, 4, …, ∞) and the negative natural numbers (-1, -2, -3, -4, …, ∞). Obviously we cannot represent the full set, because infinity is undefined. But in c++, depending on your CPU architecture and your c++ compiler, an integer is represented as a 4/8byte(32/64 bit) binary string, and later in your computer science education you will learn the way that is achieved. What this tells us is that an integer is not a magical word that makes numbers (as we perceive them) exist in a computer, but references an **encapsulation** of data(bits) in memory. Encapsulation in this context means those 4/8 bytes belong to the integer and should not be accessed by anything other than the integer. For example if I compile the following code on a 32bit architecture:

CODE1

```
3
4   int number;
5   char character;
6
```

That allocates 4 bytes for an integer object, and 1byte for a character object (characters require less memory to represent sufficiently, because the set of characters is smaller than the set of Integers). We know that there is no way that "number" can access the memory

associated with “character” and vise versa. This is encapsulation. However encapsulation in the context of class design goes beyond this, which we will see a little later on. Since int and char are native types the abstractions for them are very simple, making them the most basic objects in a program.

By now you very likely have used a class, and sort of understand what a class does/is. And if you haven’t I’ll quickly explain the motivation behind wanting a class. Keeping with the previous examples suppose you are making a program that keeps track of people for a school database. We already have an abstraction for this remember? For this example we will keep the abstraction very simple and just keep track of name, age and Student ID. Our abstraction looks something like this.

EX4

Person:
- Name
- Age
- Student ID

So how can I keep track of a database of thousands of people each of which has an associated name, age and Student ID? You might do something like this:

CODE2

```
string names[SIZE_OF_DATABASE];
int age[SIZE_OF_DATABASE];
int StudentID[SIZE_OF_DATABASE];
```

And then do a lot of work making sure that each person has the same index in every array. But that is a lot of extra work. So c++, and in fact every object oriented programming language, provides a way of keeping related values **encapsulated**. There are two ways to encapsulate data in c++. The first is with a struct, **struct** (short for structure) is a c/c++ key word that tells the compiler “everything I am going to write next should be contiguously (immediately next too each other) allocated in memory, until I say stop”. There is some extra work the compiler does so that we can later access those contiguously allocated objects, but that is not important right now.
A struct declaration looks like this:

CODE3

```
struct Person{
  string name;
  int age;
  int StudentID;
};
```

This syntax declares a new **type** called Person, this new type is defined to be represented by a string and two integers. The variables (name, age and StudentID) inside the struct are called member variables. In a struct, members are always public; meaning they can be accessed from outside the encapsulation.
The next section deals specifically with classes.

Classes

The second way of encapsulating data is with a class, classes are declared similarly to structs and mean much the same thing.

CODE4

```
class Person{
  string name;
  int age;
  int StudentID;
};
```

```
class Person{
private:
  string name;
  int age;
  int StudentID;
public:
  //more to follow
};
```

Here we see that a class declaration is almost the same as a struct declaration, however in the first class declaration, it's important to note that in a class, by default, members are private. A private member is simply a member that cannot be accessed from outside the encapsulation. In the second example you can see that the privacy is specifically stated. The syntax for this is fairly straightforward and I won't go into too much detail on that front.

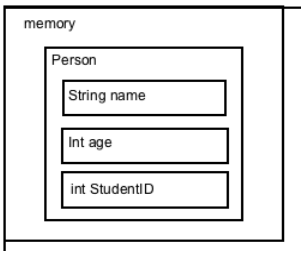
You've probably noticed that I keep talking about accessing members from inside and outside encapsulations, but I haven't really described what that is, you've probably already thought to yourself "Cool I can lump related data together, and create types! Awesome, but then what? How do I get to that data and do something with it?" Well fear not, I haven't forgotten about that, I don't want to focus on syntax too much but this is important. Once you have declared your class as I have done above you can then start using it, by declaring objects of the new type you have declared.

CODE5

```
#include <string>
#include <iostream>

using namespace std;

class Person{
public:
  string name;
  int age;
  int StudentID;

};

int main(){
  Person frank;
  frank.name = "frank";
  frank.age = 42;
  frank.StudentID = 5555556;

  cout << frank.name << endl;
  cout << frank.age << endl;
  cout << frank.StudentID << endl;

  return 0;
}
```

The code to the right shows the Person class from before, with all the members set to public.
Inside main(), an object of type Person is declared, what that means is contiguous memory has been allocated for a string object and two int objects.
EX5 gives an abstract depiction of the object in memory.

EX5



Then the members are accessed via dot notation

ObjectName.MemberName

Note that variable objects are not the only values that can be encapsulated using structs and classes. Behavior can also be encapsulated in a class. Behavior of objects is encapsulated using member functions.

CODE6

```cpp
#include <string>
#include <iostream>

using namespace std;

class Person{
private:
  string name;
  int age;
  int StudentID;
public:
  //Constructor
  Person(string n, int a, int ID): name(n), age(a),
  //Accessors
  string getName(){ return name; }
  int getAge(){ return age; }
  int getID(){ return StudentID;}

};

int main(){
  //Creating frank(an object of type Person)
  Person frank("frank", 42, 5555556);

  //Using accessors to output frank's value
  cout << frank.getName() << endl;
  cout << frank.getAge() << endl;
  cout << frank.getID() << endl;

  return 0;
}
```

This new example shows us the same class from previous examples, except with our members set to private, remember that a private member cannot be accessed with dot notation. For that reason we have written a few **accessor** functions that give us access to our member's values. This may seem a little over complicated and unnecessary now, but it's important to ensure that your objects data cannot be corrupted later. Also in the example, is a constructor, Constructors are the functions that are called when the object is created and initializes the objects members.

The sort of behavior that can be encapsulated is limited only by your ability to write functions, i.e. you can give a class just about any behavior you want, a powerful tool indeed. But, just because you can do something doesn't mean that you should, you should include appropriate and relevant behavior in your abstraction so as to separate unnecessary behavior from what you really need for your application, more on this in the next section.

Contracts

This last, but certainly not least, section deals with **contracts**. Contracts are the rules you define and **enforce**, for how your class interacts with its environment and behaves. These rules should be formal, precise and verifiable specifications for your classes interface. In essence they are an extension of the abstraction you create for your type. For example, a contract enforced in **CODE6** is that objects of type Person must be initialized (Person objects cannot be constructed without initialization parameters) because there is no defined default constructor for Person, attempting to construct a Person object without initializing it will result in a compiler error. However to be a good contract this must be specifically specified in documentation. Since this is such a small program it is sufficient to place this in a comment. On a larger project it may be more suitable to have a more readable documentation, like a PDF.

In these three files we see not only the Person abstraction realized in code, but we also see another important quality of good class design, modularity. The two files Person.hpp and Person.cpp constitute a module, they are designed generically enough that these classes can be used in many applications without modification, and the simple fact that they are in their own dedicated files gives them useful modularity. Being modular is a good thing. Here we see three contracts, the one we discussed earlier and two more. Read through the code and think about these contracts and how they are enforced. Also, I implemented an overloaded operator for stream operations. There are other operations that I could have implemented, but for the sake of brevity I did not. Think about what else could be implemented in this class. Functionality should only be included if it is necessary, and/or useful. For example a + operator would not be useful in a person class, but a ++ operator could have some value.

CODE7

Person.hpp

```cpp
//the Person class represents a Person by
//name (string)
//age (int)
//StudentID (int)

//A person must all ways be initiated,
//failure to do so will resut in a compiler Error

//a Person is outputted via stream operators
//and takes the form of an easily parsible triple
//of the form (name,age,studentID)

//a Persons members can ONLY be modified through
//mutator functions

#include <string>
#include <iostream>

class Person{
private:
  std::string name;
  int age;
  int StudentID;


public:

  Person(std::string n, int a, int ID);
  //Accessors
  std::string getName() const;
  int getAge() const;
  int getID() const;


  //mutators
  void setName( std::string );
  void setAge( int );
  void setID( int );

};

std::ostream& operator<<(std::ostream&, Person&);
```

Person.cpp

```cpp
#include <iostream>
#include <string>
#include "Person.hpp"

//MEMBER FUNCTIONS
//Constructor
Person::Person(std::string n, int a, int ID):
 name(n), age(a), StudentID(ID)
{}

//Accessors
std::string  Person::getName() const { return name; }
int Person::getAge() const { return age; }
int Person::getID() const { return StudentID;}


//Mutators
void Person::setName( std::string n)
{ name = n;}
void Person::setAge( int a )
{ age = a; }
void Person::setID( int ID )
{ StudentID = ID; }

//output
std::ostream& operator<<(std::ostream& os, Person& person )
{
  os << "("
    << person.getName()
    << ", "
    << person.getAge()
    << ", "
    << person.getID()
    <<")";
  return os;
}
```

main.cpp

```cpp
#include <string>
#include <iostream>
#include "Person.hpp"

using namespace std;


int main(){
  //Creating frank(an object of type Person)
  Person frank("frank", 42, 5555556);

  //Using contracts to output frank's value
  cout << frank<< endl;

  //Changing frank
  frank.setName("Frank");
  frank.setAge(43);
  frank.setID(0000002);

  //Frank got older and a new ID
  cout << frank << endl;
  return 0;
}
```

This paper is not a complete guide to class design, far from it, but it provides a good foundation for you to jump, full on, into object oriented programming.