Introduction to Objects

"The formation of the problem is half the solution."
— Einstein.

1 Object Oriented Languages

Object Oriented programming is not new.

- 1967 – Simula
- 1975 – Smalltalk (Squeak and Pharo)
- 1985 – C++, Objective-C
- 1995 – Java
- 2000 – C#
- 2010 – Kotlin (functional also)
- 2014 – Swift (functional also)

OO Languages–Corporate Influence

Corporate Influence on Object Oriented Programming (OOP).

- 1967 – Simula
- 1975 – Smalltalk (Xerox)
- 1985 – C++ (AT&T), Objective-C (NeXT/Apple)
- 1995 – Java (Sun)
- 2000 – C# (Microsoft)
- 2010 – Kotlin (JetBrains)
- 2014 – Swift (Apple)

OO Languages–People

- Simula – Kristen Nygaard and O.J. Dahl
- Smalltalk – Allan Kay
- C++ – Bjarne Stroustrup
- Objective-C – Brad Cox (Software ICs)
- Java – James Gosling
  James Gosling: *Simula: a personal journey*
  `http://simula67.at.ifi.uio.no/50years/index.html`
  I started using Simula in 1974 (1975?). I even did some bug fixing on the copy of the compiler that we had. It totally transformed the way I thought about software. I did use it for simulation, but I used it in other ways as well. When Smalltalk and C++ came around, neither felt to me like they measured up to Simula. And Simula had a primitive multi-threading model (via co-routines) that was ahead of its time. When it became my turn, and I was working on Java, Simula was very much on my mind.
- C# – Anders Hejlsberg (Creator of Turbo Pascal, Delphi and C#)
- Swift – Chris Lattner (Apple)

2 Objects?

What is an object?

Representation in software of some real entity.

How can an object be implemented?

Language dependent: `structs` (records), classes.

What is the difference?

`structs` are traditionally data only and access is public.
Classes include methods to manipulate underlying data.

Objects?

What is an object?

Representation in software of some real entity.

How can an object be implemented?

Language dependent: classes and/or `structs` (records).

What is the difference?

`structs` are traditionally data only and access is public.
Classes include methods to manipulate underlying data.

How are classes defined?

Classes (or pseudo-classes), in C/C++, are often defined in two files.

1. Interface (e.g., *filename.h*)
2. Implementation (e.g., *filename.cpp*)

The *interface* defines the contents and operations.

The *implementation* defines the code to manipulate the object.

3 Geometric Objects

- Circle
- Square
- Rectangle

C++ uses classes, as do many other object oriented (OO) languages. Can simulate object oriented programming (OOP) in other languages.

3.1 `struct` Technique

A circle structure:

```
struct circle
{
   double radius;
};

typedef struct circle Circle;
```

Note: `struct`s are often defined in header files when they are going to be used in multiple places.

3.1.1 Test Program

```
/*  testCircle.cpp
    C-style "object" implementation.
 */
#include <iostream>

using namespace std;

struct circle {            //  Circle definition
   double radius;
};
typedef struct circle Circle;

void SetRadius( Circle& c, double r );
void ShowRadius( Circle c );

int main()
{
    Circle c1;

    c1.radius = 2.0;
    ShowRadius( c1 );
    SetRadius( c1, 3.2 );
    ShowRadius( c1 );

    return 0;
}

void SetRadius( Circle& c, double r )
{
    c.radius = r;
}

void ShowRadius( Circle c )
{
    cout << c.radius << endl;
}
```

3.2 Class Technique

Note: the interface is defined in a separate file—`circle.h`.

```
/*  circle.h

    Circle class interface

    Bruce M. Bolden                          May 4, 1998
*/

#include <iostream>

using namespace std;

class Circle
{
public:
            //  constructors
        Circle();
        Circle( double r );

  void  SetRadius( double r );
  void  ShowRadius();

private:
  double radius;
};                  // <- Note semi-colon
```

Note: the implementation *includes* the interface file for the Circle class—`circle.h`.

```
/*  circle.cpp

    Circle class implementation

    Bruce M. Bolden                      May 4, 1998
*/

#include "circle.h"          /*  Note:  Use of "'s  */

Circle::Circle()             //  No return type?  Why?
{
    radius = 1.0;
}

Circle::Circle( double r )
{
    radius = r;
}

void Circle::SetRadius( double r )
{
    radius = r;
}

void Circle::ShowRadius()
{
    cout << radius << endl;
}
```

```
/*  TestCircle.cpp

    Bruce M. Bolden                      May 4, 1998
*/

#include <iostream>

using namespace std;

#include "circle.h"         /*  Note:  Use of "s  */


int main()
{
    Circle c1;
    Circle c2( 3.0 );

    cout << "The radius of c1 is: ";
    c1.ShowRadius();

    cout << "The radius of c2 is: ";
    c2.ShowRadius();

    c1.SetRadius( 2.1 );
    cout << "The radius of c1 is: ";
    c1.ShowRadius();

    return EXIT_SUCCESS;
}
```

3.3 How to compile?

```
g++ TestCircle.cpp circle.cpp
```

What about `circle.h`?

Processed (included) when `TestCircle.cpp` and `circle.cpp` are compiled.

3.4 Example: Dice

Just as we created a circle object class, we can create a dice class.

Design Issues:

- What are the properties of a die?
- What are the operations of a die?

3.4.1 Interface

Note: the interface is defined in a separate file—`dice.h`.

```
/*  dice.h

    Dice class interface

    Bruce M. Bolden       July 5, 1998
*/

class Dice
{
public:
        Dice();
        Dice( int nFaces );

  int   Roll();

private:
  int   faces;
};
```

3.4.2 Implementation

Note: the implementation *includes* the interface file for the dice class—`dice.h`.

```
/*  dice.cpp

    Dice class implementation

    Bruce M. Bolden      July 5, 1998
*/

#include <iostream>

using namespace std;

#include "dice.h"

Dice::Dice()
{
    faces = 6;
}

Dice::Dice( int nFaces )
{
    faces = nFaces;
}

int Dice::Roll()
{
    return  1 + rand() % faces;
}
```

3.4.3 Test Program

```
/*  testDice.cpp

    Test the dice class.
*/

#include <iostream>

using namespace std;

#include "dice.h"

int main()
{
    int  i, nRolls = 6;

        //  dice objects
    Dice d1;
    Dice d2( 12 );

    cout << "Roll of d1: " << endl;
    for( i = 0 ; i < nRolls ; ++i )
        cout << d1.Roll() << endl;

    cout << "\nRolling d2:" << endl;
    for( i = 0 ; i < nRolls ; ++i )
        cout << d2.Roll() << endl;

    return EXIT_SUCCESS;
}
```

3.4.4 Output

```
Roll of d1:
3
5
4
2
2
6

Rolling d2:
7
4
1
7
2
12
```

3.5 Example: Craps revised

```
/*  craps.cpp

    Revised craps program.  Dice are objects.

    Derived from:
    C++ How to Program, Deitel & Deitel, 1998
*/

#include <iostream>
#include <cstdlib>
#include <time.h>

using namespace std;

#include "dice.h"

    //  global objects
Dice   die1, die2;

    //  prototypes
int RollDice();
int Original_RollDice();
```

```
int main()
{
    int sum, myPoint;

    enum   Status { CONTINUE, WON, LOST };
    Status  gameStatus;

        //  Initialize random number generator
    srand( time(NULL) );

    sum = RollDice();           //  first throw of dice

    switch( sum ) {
    case  7:                    //  win on first roll
    case 11:
        gameStatus = WON;
        break;

    case  2:                    //  lose on first roll
    case  3:
    case 12:
        gameStatus = LOST;
        break;

    default:                    //  remember point
        gameStatus = CONTINUE;
        myPoint = sum;
        cout << " Point is " << myPoint << endl;
        break;
    }
```

```
        //  keep rolling
    while( gameStatus == CONTINUE )
    {
        sum = RollDice();

        if( sum == myPoint )   //  win by making point
            gameStatus = WON;
        else if( sum == 7 )
            gameStatus = LOST;
    }

        //  show results
    if( gameStatus == WON )
    {
        cout << " Player wins" << endl;
    }
    else
    {
        cout << " Player loses" << endl;
    }

    return EXIT_SUCCESS;
}
```

```
int RollDice()
{
    int d1, d2, sum;

    d1  = die1.Roll();
    d2  = die2.Roll();
    sum = d1 + d2;
    //sum = die1.Roll() + die2.Roll();

    cout << "Player rolled " << d1 << " + " << d2;
    cout << " = " << sum << endl;

    return sum;
}
```

```
int Original_RollDice()
{
    int die1, die2, sum;

    die1 = 1 + rand() % 6;
    die2 = 1 + rand() % 6;
    sum  = die1 + die2;

    cout << "Player rolled " << die1 << " + " << die2;
    cout << " = " << sum << endl;

    return sum;
}
```

Die, C, die! 5 reasons to UN-learn C.

`http://blogs.zdnet.com/Burnette/?p=208`

`November 28, 2006`
`Die, C, die! 5 reasons to UN-learn C.`
`Posted by Ed Burnette @ 12:01 am`

I've been programming in C for over 20 years now. I've written C compilers, C debuggers, other languages, games, clients, servers, you name it. Dog-eared editions of K&R and Steele decorate my shelves. So I know C. And yet, I'm sick of it. SICK.

So it was with some trepidation that I read a blog on why every programmer should learn C. Turns out it's good for a laugh if you're a professional developer, though the author probably didn't intend it that way. This rebuttal makes a bit more sense, but still doesn't capture the essence of why C should go the way of the dodo. So let me turn it around. Here are 5 reasons why developers who know and use C now should not just use something else, but UN-learn all the bad things they learned in C.

1. Memory allocation. I could write a whole article just on this one. A book. Maybe a small wing of the library. Memory allocation and deallocation is the bane of my existence. Either you allocate too little and write off the end, or too much and waste it. Then there's the question of whether to zero it or leave it uninitialized. But freeing memory is the worst. Entire toolkits have been written to help you make sure you have freed every little bit you allocated, never use it after freeing, and God forbid, never free it twice. To add insult to injury, allocations and frees are slow in C, very slow. I don't want to even think

about all the special cases I've had to put in to *avoid* memory allocation and use stack or pre-allocated structure space if the problem size fit. Well, I've got better things to worry about. Whoever invented garbage collection should win a Nobel.

2. Multi-threading. I used to like C, really. Until I started to develop and maintain multi-threaded servers with it. C doesn't help you at all with protecting data from access by conflicting threads. Every intuition you had from single-threaded days is wrong. At least Java has the `synchronized` keyword, and a documented (but weird) memory model, but even that falls apart on massively parallel machines unless you use the new `javax.concurrent stuff`. Flashback – in C: 1 week standing up (true story) in a data center debugging a deadlock problem in a simulated production environment. In Java: Ctrl+Break! Ahhh.

3. Pointers. Pointers are insidiously evil; there's just no polite way to say it. Months of my life are just gone from debugging problems with wild pointers. I used to go for all the tricks, such as incomprehensible casts and unions and offsetof and reusing the last couple of bits for flags, and all that. It's just not worth it. Statically typed references are your friend.

4. Premature optimization. Speaking of tricks, have you ever wasted any brain cells wondering if `*p++` was faster than `p[i]`? Have you spent time trying to do shifts instead of multiplies, or reversing for loops to try and make them run faster? Agonized over the speed of passing parameters as opposed to filling in a structure and passing that? STOP IT! Algorithms are the key to speed, and developer productivity is the key to algorithms. Get the idea that you can make your program any better or faster with little tweaks out of your head. Yeah, there

are a few cases where maybe no, just don't go there.

5. Tests. What's your favorite C unit testing tool? Umm.. can't think of one? Unit testing must not be important then, right? Or too much trouble. Hard to keep up to date. Waste of time. You could spend that time shaving .001% off your execution time. Or debugging that problem that only occurs with 100 simultaneous users, in the data center, on an optimized image with no symbols.

I could go on, but 5 is enough for now; I feel better already. C was wonderful ... in 1984. It amazes me that new code is being written in C, even today. C++ is only marginally better if you ask me. If you want to learn something old, try Forth, Lisp, or APL. At least those can teach you some different and elegant ways of thinking about programming.