UNIVERSITY OF OREGON

CIS 507:

```
   __  __      _                          __   ______    ________
  / / / /___  (_)  __   ____ _____  ____/ /  / ____/  _/_/ ____/__    __
 / / / / __ \/ / |/_/  / __ `/ __ \/ __  /  / /     _/_// /  __/ /___/ /_
/ /_/ / / / / />  <   / /_/ / / / / /_/ /  / /___ _/_/ / /__/_  __/_  __/
\____/_/ /_/_/_/|_|   \__,_/_/ /_/\__,_/   \____//_/   \____//_/   /_/
```

Lecture 9:
how C++ works under the covers, and also exceptions

November 20th, 2018 Hank Childs, University of Oregon

Function Pointers

Function Pointers

- Idea:
  - You have a pointer to a function
  - This pointer can change based on circumstance
  - When you call the function pointer, it is like calling a known function

Function Pointer Example

```
128-223-223-72-wireless:cli hank$ cat function_ptr.c
#include <stdio.h>
int doubler(int x) { return 2*x; }
int tripler(int x) { return 3*x; }
int main()
{
    int (*multiplier)(int);
    multiplier = doubler;
    printf("Multiplier of 3 = %d\n", multiplier(3));
    multiplier = tripler;
    printf("Multiplier of 3 = %d\n", multiplier(3));
}
128-223-223-72-wireless:cli hank$ gcc function_ptr.c
128-223-223-72-wireless:cli hank$ ./a.out
Multiplier of 3 = 6
Multiplier of 3 = 9
```

Function Pointers vs Conditionals

```
128-223-223-72-wireless:cli hank$ cat function_ptr2.c
#include <stdio.h>
int doubler(int x) { return 2*x; }
int tripler(int x) { return 3*x; }
int main()
{
    int (*multiplier)(int);
    int condition = 1;

    if (condition)
       multiplier = doubler;
    else
       multiplier = doubler;

    printf("Multiplier of 3 = %d\n", multiplier(3));
}
```

```
#include <stdio.h>
int doubler(int x) { return 2*x; }
int tripler(int x) { return 3*x; }
int main()
{
    int val;

    if (condition)
       val = doubler(3);
    else
       val = tripler(3);

    printf("Multiplier of 3 = %d\n", val);
}
```

What are the pros and cons of each approach?

Function Pointer Example #2

```
128-223-223-72-wireless:cli hank$ cat array_fp.c
#include <stdio.h>
void doubler(int *X) { X[0]*=2; X[1]*=2; };
void tripler(int *X) { X[0]*=3; X[1]*=3; };
int main()
{
    void (*multiplier)(int *);
    int A[2] = { 2, 3 };
    multiplier = doubler;
    multiplier(A);
    printf("Multiplier of 3 = %d, %d\n", A[0], A[1]);
    multiplier = tripler;
    multiplier(A);
    printf("Multiplier of 3 = %d, %d\n", A[0], A[1]);
}
128-223-223-72-wireless:cli hank$ gcc array_fp.c
128-223-223-72-wireless:cli hank$ ./a.out
```

Function pointer

Part of function signature

Don't be scared of extra '*'s … they just come about because of pointers in the arguments or return values.

Simple-to-Exotic Function Pointer Declarations

```
void (*foo)(void);
void (*foo)(int **, char ***);
char ** (*foo)(int **, void (*)(int));
```

These sometimes come up on interviews.

Callbacks

- Callbacks: function that is called when a condition is met
  - Commonly used when interfacing between modules that were developed separately.
  - … libraries use callbacks and developers who use the libraries “register” callbacks.

Callback example

```
128-223-223-72-wireless:callback hank$ cat mylog.h
void RegisterErrorHandler(void (*eh)(char *));
double mylogarithm(double x);
```

```
128-223-223-72-wireless:callback hank$ cat mylog.c
#include <mylog.h>

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* NULL is an invalid memory location.
 * Useful for setting to something known, rather than
   leaving uninitialized */
void (*error_handler)(char *) = NULL;

void RegisterErrorHandler(void (*eh)(char *))
{
    error_handler = eh;
}

void Error(char *msg)
{
    if (error_handler != NULL)
        error_handler(msg);
}

double mylogarithm(double x)
{
    if (x <= 0)
    {
        char msg[1024];
        sprintf(msg, "Logarithm of a negative number: %f !!", x);
        Error(msg);
        return 0;
    }

    return log(x);
}
```

Callback example

```
128-223-223-72-wireless:callback hank$ cat program.c
#include <mylog.h>
#include <stdio.h>

FILE *F1 = NULL;
void HanksErrorHandler(char *msg)
{
    if (F1 == NULL)
    {
        F1 = fopen("error", "w");
    }
    fprintf(F1, "Error: %s\n", msg);
}

int main()
{
    RegisterErrorHandler(HanksErrorHandler);

    mylogarithm(3);
    mylogarithm(0);
    mylogarithm(-2);
    mylogarithm(5);
    if (F1 != NULL)
       fclose(F1);
}
128-223-223-72-wireless:callback hank$
128-223-223-72-wireless:callback hank$ ./program
128-223-223-72-wireless:callback hank$
128-223-223-72-wireless:callback hank$ cat error
Error: Logarithm of a negative number: 0.000000 !!
Error: Logarithm of a negative number: -2.000000 !!
128-223-223-72-wireless:callback hank$
```

How C++ does OOP under the covers

"this": pointer to current object

- From within any struct's method, you can refer to the current object using "this"

```
TallyCounter::TallyCounter(int c)
{
     count = c;
}

      <------>

TallyCounter::TallyCounter(int c)
{
     this->count = c;
}
```

How methods work under the covers (1/4)

```
class  MyIntClass
{
  public:
                  MyIntClass(int x) { myInt = x; };

    friend void   FriendIncrementFunction(MyIntClass *);
    int           GetMyInt() { return myInt; };

  protected:
    int           myInt;
};

void
FriendIncrementFunction(MyIntClass *mic)
{
    mic->myInt++;
}

int main()
{
    MyIntClass MIC(12);
    FriendIncrementFunction(&MIC);
    FriendIncrementFunction(&MIC);
    cout << "My int is " << MIC.GetMyInt() << endl;
}
```

```
fawcett:330 childs$ g++ this.C
fawcett:330 childs$ ./a.out
My int is 14
fawcett:330 childs$
```

How methods work under the covers (2/4)

```
class  MyIntClass
{
  public:
                  MyIntClass(int x) { myInt = x; };

    friend void   FriendIncrementFunction(MyIntClass *);
    int           GetMyInt() { return myInt; };

  protected:
    int           myInt;
};

void
FriendIncrementFunction(MyIntClass *mic)
{
    mic->myInt++;
}

int main()
{
    MyIntClass MIC(12);
    FriendIncrementFunction(&MIC);
    FriendIncrementFunction(&MIC);
    cout << "My int is " << MIC.GetMyInt() << endl;
}
```

Addr.	Variable	Value
0x8000	MIC/myInt	12

Addr.	Variable	Value
0x8000	MIC/myInt	12
0x8004	mic	0x8000

How methods work under the covers (3/4)

```
class  MyIntClass
{
  public:
                   MyIntClass(int x) { myInt = x; };

    friend void    FriendIncrementFunction(MyIntClass *);
    void           IncrementMethod(void);
    int            GetMyInt() { return myInt; };

  protected:
    int            myInt;
};

void
FriendIncrementFunction(MyIntClass *mic)
{
    mic->myInt++;
}

void
MyIntClass::IncrementMethod(void)
{
    this->myInt++;
}

int main()
{
    MyIntClass MIC(12);
    FriendIncrementFunction(&MIC);
    MIC.IncrementMethod();
    cout << "My int is " << MIC.GetMyInt() << endl;
}
```

```
fawcett:330 childs$ g++ this.C
fawcett:330 childs$ ./a.out
My int is 14
fawcett:330 childs$
```

How methods work under the covers (4/4)

```
class  MyIntClass
{
}

v
FriendIncrementFunction(MyIntClass *mic)
{
    mic->myInt++;
}

void
MyIntClass::IncrementMethod(void)
{
    this->myInt++;
}

int main()
{
    MyIntClass MIC(12);
    FriendIncrementFunction(&MIC);
    MIC.IncrementMethod();
    cout << "My int is " << MIC.GetMyInt() << endl;
}
```

The compiler secretly slips “this” onto the stack whenever you make a method call.

It also automatically changes “myInt” to this->myInt in methods.

Addr.	Variable	Value
0x8000	MIC/myInt	12

Addr.	Variable	Value
0x8000	MIC/myInt	12
0x8004	mic	0x8000

Addr.	Variable	Value
0x8000	MIC/myInt	13
0x8004	this	0x8000

Virtual Function Tables

Virtual functions

- Virtual function: function defined in the base type, but can be re-defined in derived type.
- When you call a virtual function, you get the version defined by the derived type

Virtual functions: example

```
128-223-223-72-wireless:330 hank$ cat virtual.C
#include <stdio.h>

struct SimpleID
{
   int id;
   virtual int GetIdentifier() { return id; };
};

struct ComplexID : SimpleID
{
   int extraId;
   virtual int GetIdentifier() { return extraId*128+id; };
};

int main()
{
   ComplexID cid;
   cid.id = 3;
   cid.extraId = 3;
   printf("ID = %d\n", cid.GetIdentifier());
}
128-223-223-72-wireless:330 hank$ g++ virtual.C
128-223-223-72-wireless:330 hank$ ./a.out
ID = 387
```

Picking the right virtual function

```
class A
{
  public:
    virtual const char *GetType() { return "A"; };
};

class B : public A
{
  public:
    virtual const char *GetType() { return "B"; };
};

int main()
{
    A a;
    B b;

    cout << "a is " << a.GetType() << endl;
    cout << "b is " << b.GetType() << endl;
}

fawcett:330 childs$ g++ virtual.C
fawcett:330 childs$ ./a.out
```

??????

It seems like the compiler should be able to figure this out ...
it knows that a is of type A
and
it knows that b is of type B

Picking the right virtual function

```
class A
{
  public:
    virtual const char *GetType() { return "A"; };
};

class B : public A
{
  public:
    virtual const char *GetType() { return "B"; };
};

void
ClassPrinter(A *ptrToA)
{
    cout << "ptr points to a " << ptrToA->GetType() << endl;
}

int main()
{
    A a;
    B b;

    ClassPrinter(&a);
    ClassPrinter(&b);
}

fawcett:330 childs$ g++ virtual2.C
fawcett:330 childs$ ./a.out
```

??????

So how to does the compiler know?

How does it get “B” for “b” and “A” for “a”?

Virtual Function Table

- Let C be a class and X be an instance of C.
- Let C have 3 virtual functions & 4 non-virtual functions
- C has a hidden data member called the "virtual function table"
- This table has 3 rows
  - Each row has the correct definition of the virtual function to call for a "C".
- When you call a virtual function, this table is consulted to locate the correct definition.

Showing the existence of the virtual function pointer with sizeof()

```
class A
{
  public:
    virtual
};

class B : pub
{
  public:
    virtual
};

class C
{
  public:
    const char *GetType() { return "C"; };
};

int main()
{
    A a;
    B b;

    cout << "Size of A is " << sizeof(A) << endl;
    cout << "Size of a pointer is " << sizeof(int *) << endl;
    cout << "Size of C is " << sizeof(C) << endl;
}
```

empty objects have size of 1? why?!?

Answer: so every object has a unique address.

```
fawcett:330 childs$ ./a.out
Size of A is 8
Size of a pointer is 8
Size of C is 1
```

what will this print?

Virtual Function Table

- Let C be a class and X be an instance of C.
- Let C have 3 virtual functions & 4 non-virtual functions
- Let D be a class that inherits from C and Y be an instance of D.
  - Let D add a new virtual function
- D’s virtual function table has 4 rows
  - Each row has the correct definition of the virtual function to call for a “D”.

More notes on virtual function tables

- There is one instance of a virtual function table for each class
  - Each instance of a class shares the same virtual function table
- Easy to overwrite (i.e., with a memory error)
  - And then all your virtual function calls will be corrupted
  - Don’t do this! ;)

Virtual function table: example

CIS 330: Project #2C
Assigned: April 17th, 2014
Due April 24th, 2014
(which means submitted by 6am on April 25th, 2014)
Worth 6% of your grade

Please read this entire prompt!

Assignment: You will implement subtypes with C.

1) Make a union called ShapeUnion with the three types (Circle, Rectangle, Triangle).
2) Make a struct called FunctionTable that has pointers to functions.
3) Make an enum called ShapeType that identifies the three types.
4) Make a struct called Shape that has a ShapeUnion, a ShapeType, and a FunctionTable.
5) Modify your 9 functions to deal with Shapes.
6) Integrate with the new driver function. Test that it produces the correct output.

Virtual function table: example

```
class Shape
{
    virtual double GetArea() = 0;
    virtual void   GetBoundingBox(double *) = 0;
};

class Rectangle : public Shape
{
  public:
                   Rectangle(double, double, double, double);
    virtual double GetArea();
    virtual void   GetBoundingBox(double *);
  protected:
    double minX, maxX, minY, maxY;
};

class Triangle : public Shape
{
  public:
                   Triangle(double, double, double, double);
    virtual double GetArea();
    virtual void   GetBoundingBox(double *);
  protected:
    double pt1X, pt2X, minY, maxY;
};
```

Questions

- What does the virtual function table look like for a Shape?

```
typedef struct
{
    double (*GetArea)(Shape *);
    void   (*GetBoundingBox)(Shape *, double *);
} VirtualFunctionTable;
```

- What does Shape's virtual function table look like?
  – Trick question: Shape can't be instantiated, precisely because you can't make a virtual function table
    - abstract type due to pure virtual functions

Questions

- What is the virtual function table for Rectangle?

```
c->ft.GetArea = GetRectangleArea;
c->ft.GetBoundingBox = GetRectangleBoundingBox;
```

- (this is a code fragment from my 2C solution)

Calling a virtual function

- Let X be an instance of class C.
- Assume you want to call the 4th virtual function
- Let the arguments to the virtual function be an integer Y and a float Z.
- Then call:

(X.vptr[3])(&X, Y, Z);

The 4th virtual function has index 3 (0-indexing)

The pointer to the virtual function pointer (often called a vptr) is a data member of X

Secretly pass "this" as first argument to method

Inheritance and Virtual Function Tables

```
class A
{
  public:
```

> This whole scheme gets much harder with multiple inheritance, and you have to carry around multiple virtual function tables.

```
    virtual void Foo2();
};
```

```
class C : public B
{
  public:
    virtual void Foo1();
    virtual void Foo2();
    virtual void Foo3();
};
```

A	
	Location of Foo1

	Location of Foo1
	Location of Foo2

C	
Foo1	Location of Foo1
Foo2	Location of Foo2
Foo3	Location of Foo3

Same as B's
This is how you can treat a C as a B

Virtual Function Table: Summary

- Virtual functions require machinery to ensure the correct form of a virtual function is called
- This is implemented through a virtual function table
- Every instance of a class that has virtual functions has a pointer to its class's virtual function table
- The virtual function is called via following pointers
  - Performance issue

Now show Project 2D in C++

- Comment:
  - C/C++ great because of performance
  - Performance partially comes because of a philosophy of not adding "magic" to make programmer's life easier
  - C has very little pixie dust sprinkled in
    - Exception: '\0' to terminate strings
  - C++ has more
    - Hopefully this will demystify one of those things (virtual functions)

vptr.C

```
fawcett:vptr childs$ cat vptr.C
#include <iostream>
using std::cerr;
using std::endl;

class Shape
{
  public:
     int s;
     virtual double GetArea() = 0;
     virtual void   GetBoundingBox(double *) = 0;
};

class Triangle : public Shape
{
  public:
     virtual double GetArea() { cerr << "In GetArea for Triangle" << endl; return 1;};
     virtual void GetBoundingBox(double *) { cerr << "In GetBBox for Triangle" << endl; };
};

class Rectangle : public Shape
{
  public:
     virtual double GetArea() { cerr << "In GetArea for Rectangle" << endl; return 2; };
     virtual void GetBoundingBox(double *) { cerr << "In GetBBox for Rectangle" << endl; };
};

struct VirtualFunctionTable
{
     double (*GetArea)(Shape *);
     void (*GetBoundingBox)(Shape *, double *);
};


int main()
{
    Rectangle r;
    cerr << "Size of rectangle is " << sizeof(r) << endl;

    VirtualFunctionTable *vft = *((VirtualFunctionTable**)&r);
    cerr << "Vptr = " << vft << endl;
    double d = vft->GetArea(&r);
    cerr << "Value = " << d << endl;

    double bbox[4];
    vft->GetBoundingBox(&r, bbox);
}
```

Exceptions

Exceptions

- C++ mechanism for handling error conditions
- Three new keywords for exceptions
  - try: code that you “try” to execute and hope there is no exception
  - throw: how you invoke an exception
  - catch: catch an exception … handle the exception and resume normal execution

Exceptions

```
fawcett:330 childs$ cat exceptions.C
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    try
    {
        cout << "About to throw 105" << endl;
        throw 105;
        cout << "Done throwing 105" << endl;
    }
    catch (int &theInt)
    {
        cout << "Caught an int: " << theInt << endl;
    }
}
fawcett:330 childs$ g++ exceptions.C
```

Exceptions: catching multiple types

```
fawcett:330 childs$ cat exceptions2.C
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    try
    {
         cout << "About to throw 105" << endl;
         throw 105;
         cout << "Done throwing 105" << endl;
    }
    catch (int &theInt)
    {
         cout << "Caught an int: " << theInt << endl;
    }
    catch (float &theFloat)
    {
         cout << "Caught a float: " << theFloat << endl;
    }
}
fawcett:330 childs$ g++ exceptions2.C
fawcett:330 childs$ ./a.out
About to throw 105
Caught an int: 105
```

Exceptions: catching multiple types

```
fawcett:330 childs$ cat exceptions3.C
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    try
    {
         cout << "About to throw 10.5" << endl;
         throw 10.5;
         cout << "Done throwing 10.5" << endl;
    }
    catch (int &theInt)
    {
         cout << "Caught an int: " << theInt << endl;
    }
    catch (float &theFloat)
    {
         cout << "Caught a float: " << theFloat << endl;
    }
}
fawcett:330 childs$ g++ exceptions3.C
fawcett:330 childs$ ./a.out
About to throw 10.5
terminate called after throwing an instance of 'double'
Abort trap
```

Exceptions: catching multiple types

```
fawcett:330 childs$ cat exceptions4.C
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    try
    {
        cout << "About to throw 10.5" << endl;
        throw 10.5;
        cout << "Done throwing 10.5" << endl;
    }
    catch (int &theInt)
    {
        cout << "Caught an int: " << theInt << endl;
    }
    catch (float &theFloat)
    {
        cout << "Caught a float: " << theFloat << endl;
    }
    catch (double &theDouble)
    {
        cout << "Caught a double: " << theDouble << endl;
    }
}
```

```
fawcett:330 childs$ g++ exceptions4.C
fawcett:330 childs$ ./a.out
About to throw 10.5
Caught a double: 10.5
fawcett:330 childs$
```

Exceptions: throwing/catching complex types

```
class MyExceptionType  { };

class MemoryException : public MyExceptionType {};
class FailedAllocationException : public MemoryException {};
class NULLPointerException : public MemoryException {};

class FloatingPointException : public MyExceptionType {};
class DivideByZeroException : public FloatingPointException {};
class OverflowException : public FloatingPointException {};
```

```
void Foo();

int main()
{
    try
    {
        Foo();
    }
    catch (MemoryException &e)
    {
        cout << "I give up" << endl;
    }
    catch (OverflowException &e)
    {
        cout << "I think it is OK" << endl;
    }
    catch (DivideByZeroException &e)
    {
        cout << "The answer is bogus" << endl;
    }
}
```

Exceptions: cleaning up before you return

```
void Foo(int *arr);

int *
Foo2(void)
{
    int  *arr = new int[1000];
    try
    {
        Foo(arr);
    }
    catch (MyExceptionType &e)
    {
         delete [] arr;
         return NULL;
    }

    return arr;
}
```

Exceptions: re-throwing

```
void Foo(int *arr);

int *
Foo2(void)
{
    int  *arr = new int[1000];
    try
    {
        Foo(arr);
    }
    catch (MyExceptionType &e)
    {
        delete [] arr;
        throw e;
    }

    return arr;
}
```

Exceptions: catch and re-throw anything

```
void Foo(int *arr);

int *
Foo2(void)
{
    int  *arr = new int[1000];
    try
    {
        Foo(arr);
    }
    catch (...)
    {
         delete [] arr;
         throw;
    }

    return arr;
}
```

Exceptions: declaring the exception types you can throw

```
int *
MyIntArrayMemoryAllocator(int num) throw(FloatingPointException)
{
    int  *arr = new int[num];
    if (arr == NULL)
        throw DivideByZeroException();

    return arr;
}
```

Exceptions: declaring the exception types you can throw ... not all it is cracked up to be

```
int *
MyIntArrayMemoryAllocator(int num) throw(FloatingPointException)
{
    int  *arr = new int[num];
    if (arr == NULL)
        throw MemoryException();

    return arr;
}
```

This will compile ... compiler can only enforce this as a run-time thing.

As a result, this is mostly unused (I had to read up on it)

But: "standard" exceptions have a "throw" in their declaration.

std::exception

- c++ provides a base type called "std::exception"
- It provides a method called "what"

```
// using standard exceptions
#include <iostream>
#include <exception>
using namespace std;

class myexception: public exception
{
  virtual const char* what() const throw()
  {
    return "My exception happened";
  }
} myex;

int main () {
  try
  {
    throw myex;
  }
  catch (exception& e)
  {
    cout << e.what() << '\n';
  }
  return 0;
}
```

Source: cplusplus.com

Exceptions generator by C++ standard library

exception	description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when it fails in a dynamic cast
bad_exception	thrown by certain dynamic exception specifiers
bad_typeid	thrown by typeid
bad_function_call	thrown by empty function objects
bad_weak_ptr	thrown by shared_ptr when passed a bad weak_ptr

Source: cplusplus.com

3F

Project 3F in a nutshell

- Logging:
  - infrastructure for logging
  - making your data flow code use that infrastructure
- Exceptions:
  - infrastructure for exceptions
  - making your data flow code use that infrastructure

The webpage has a head start at the infrastructure pieces for you.

Warning about 3F

- My driver program only tests a few exception conditions
- Your stress tests later will test a lot more.
  - Be thorough, even if I’m not testing it

3F: warning

- 3F will almost certainly crash your code
  - It uses your modules wrong!
- You will need to figure out why, and add exceptions
  - gdb will be helpful

Bonus Material

Operator Precedence

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13[note 1]	?:	Ternary conditional[note 2]	Right-to-Left
14	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Source: http://en.cppreference.com/w/c/language/operator_precedence

Unions

- Union: special data type
  - store many different memory types in one memory location

```
typedef union
{
    float x;
    int   y;
    char  z[4];
} cis330_union;
```

When dealing with this union, you can treat it as a float, as an int, or as 4 characters.

This data structure has 4 bytes

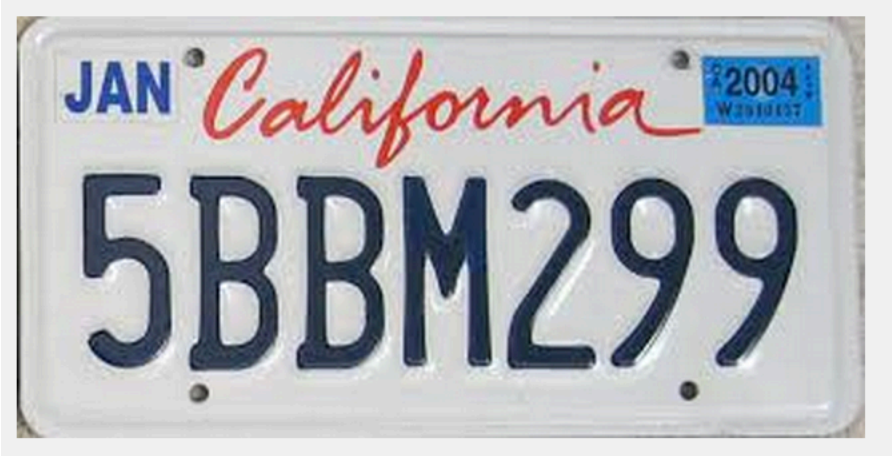
Unions

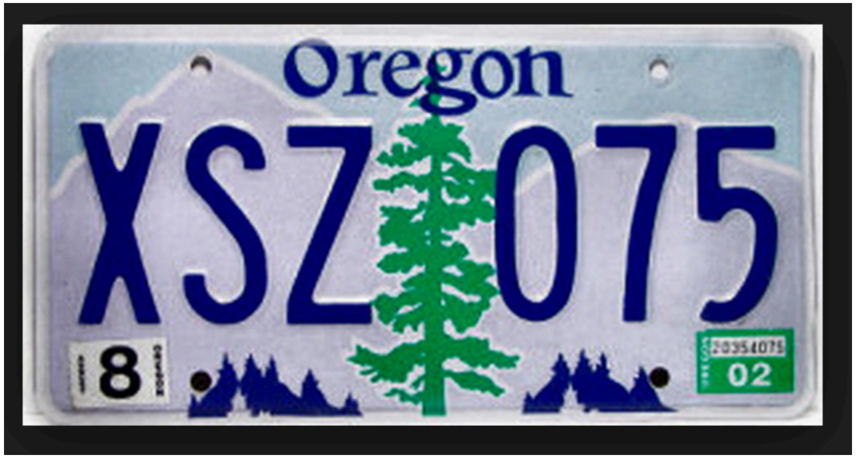
```
128-223-223-72-wireless:330 hank$ cat union.c
#include <stdio.h>

typedef union
{
    float x;
    int   y;
    char  z[4];
} cis330_union;


int main()
{
    cis330_union u;
    u.x = 3.5;  /* u.x is 3.5,     u.y and u.z are not meaningful */
    u.y = 3;    /* u.y is 3,   now u.x and u.z are not meaningful */
    printf("As u.x = %f, as u.y = %d\n", u.x, u.y);
}
128-223-223-72-wireless:330 hank$ gcc union.c
128-223-223-72-wireless:330 hank$ ./a.out
As u.x = 0.000000, as u.y = 3
```

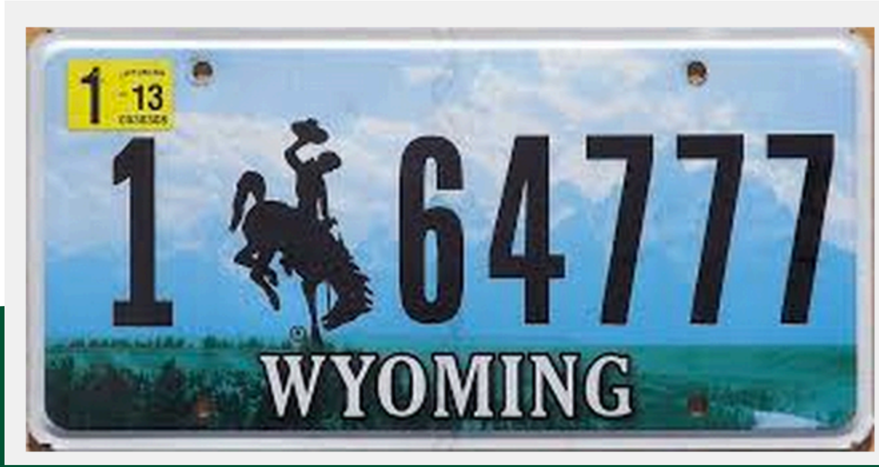
Why are unions useful?

Unions Example

```
typedef struct
{
    int firstNum;
    char letters[3];
    int endNums[3];
} CA_LICENSE_PLATE;

typedef struct
{
    char  letters[3];
    int   nums[3];
} OR_LICENSE_PLATE;

typedef struct
{
    int   nums[6];
} WY_LICENSE_PLATE;

typedef union
{
    CA_LICENSE_PLATE ca;
    OR_LICENSE_PLATE or;
    WY_LICENSE_PLATE wy;
} LicensePlate;
```







Unions Example

```
typedef struct
{
   int firstNum;
   char letters[3];
   int endNums[3];
} CA_LICENSE_PLATE;

typedef struct
{
   char  letters[3];
   int   nums[3];
} OR_LICENSE_PLATE;

typedef struct
{
   int   nums[6];
} WY_LICENSE_PLATE;

typedef union
{
   CA_LICENSE_PLATE ca;
   OR_LICENSE_PLATE or;
   WY_LICENSE_PLATE wy;
} LicensePlate;
```

```
typedef enum
{
   CA,
   OR,
   WY
} US_State;

typedef struct
{
   char *carMake;
   char *carModel;
   US_State  state;
   LicensePlate lp;
} CarInfo;

int main()
{
    CarInfo c;
    c.carMake = "Chevrolet";
    c.carModel = "Camaro";
    c.state = OR;
    c.lp.or.letters[0] = 'X';
    c.lp.or.letters[1] = 'S';
    c.lp.or.letters[2] = 'Z';
    c.lp.or.nums[0] = 0;
    c.lp.or.nums[1] = 7;
    c.lp.or.nums[2] = 5;
}
```