# Administration

- Portfolio marking is starting now
  - I’ve been either on annual leave and/or ill since submission...
- Programming tests today and tomorrow
  - Same setup as previous tests
- There is lab timetabled after, but nothing to do yet
  - Sahbi will be available for general programming questions
- Lab + portfolio start next week
  - Rewritten a bit to create direct and explicit link
  - Doing lab directly helps to complete portfolio

# What you’ve seen

- Classic “procedural” programming
    - Types and declarations
    - Control structures and operators
    - Functions and lifetimes

- Pointers and linked-data structures
    - Pointers and pointer operations
    - Dynamic memory allocation
    - Linked data-structures

# What you can do right now

- Use basic system data-types
- Design your own new data-types
- Write code for embedded systems
- Write command-line programs
- Write an operating system (sort of...)

# What comes next

Improving efficiency and managing complexity

- *Efficiency*: how fast can you create a *working* solution
- *Complexity*: how do you split problems up and collaborate?

Most of this will focus on creating and using ***abstractions***

# Abstractions

Abstractions allow us to:

- simplify complex systems
- communicate with others
- re-use the work of others
- share our own insight/work with others

# Abstractions: simplification

We don’t really *need* most mathematical objects

$$\cos(x)$$

$$\frac{df(x)}{dx}$$

$$F(\theta)$$

# Abstractions: simplification

We don’t really *need* most mathematical objects

$$\cos(x) = \mathrm{abs}(e^{ix})$$

$$\frac{df(x)}{dx} = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}$$

$$F(\theta) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x\theta} dx$$

# Abstractions: freedom

One abstraction could have many implementations

$$\cos(x) = \text{abs}\left(e^{ix}\right)$$

$$= \sum_{i=0}^{\infty} \frac{-1^n x^{2n}}{(2n)!}$$

$$= \sin(x + \pi/2)$$

# Abstractions : programming

Built-in types as abstractions:

A type is a set of possible values plus
a set of operations on that set

A built-in type comes as part of the language

We have used `int` without worrying about the details

How big is an `int`?

How is x+y implemented?

# Abstractions : programming

Functions as abstractions:

A function maps a set of input values
to a set of output values

We have used functions `to_string` and `sin`:

What algorithm does `to_string` use?

Is `sin` calculated in hardware or software?

# Abstractions : programming

Handle APIs as abstractions (e.g. struct + functions)

APIs allow the user to create new types and a set of associated operations

or

to add functionality to existing types

Operating systems use this approach extensively

# Abstractions : objects

***Functions*** provide computation without state

***Structs*** provide state without computation

Functions + structs provide state + computation
***Objects*** provide state + computation

We can capture many useful APIs with objects

# A brief API case study

# API Design: modelling the world

We often use software to represent “things”

- Circuits, maths, images, sound, controllers, ...

Lot’s of things combine computation and state

- Things can change state independently
  - A clock measuring the current time
  - A network connection where data sometimes arrives
- Things can change state due to computation
  - Adding something to a container
  - Changing a pixel in an image

# API Design : Requirements

Create an API for describing digital logic circuits

Required operations:

1. Create “and”, “or” and “not” gates
2. Connect gates together
3. Print the circuit out to cout

# API Design : Representing a circuit

*Create an API for describing digital logic circuits*

```
// There is some struct that represents a circuit
struct Circuit;

// Creates a new instance of a circuit
Circuit *create();

// Destroys an existing instance of a circuit
void destroy(Circuit *c);
```

# API Design : Adding Gates

1. *Create “and”, “or” and “not” gates*

```
struct Circuit;
```

# API Design : Adding Gates V1

1. *Create "and", "or" and "not" gates*

```
struct Circuit;

// Other structs to represent gates
struct AndGate;
struct OrGate;
struct NotGate;

// Functions to create a gate within a circuit
AndGate *add_and_gate(Circuit *c);
OrGate  *add_or_gate  (Circuit *c);
NotGate *add_not_gate (Circuit *c);
```

# API Design : Adding Gates V2

1. *Create "and", "or" and "not" gates*

```
struct Circuit;

// Struct to represent all gates
struct Gate;



// Function to create a gate within a circuit
//   gate_type : "and", "or", "not"
Gate *add_gate(Circuit *c, string gate_type);
```

# API Design : Adding Gates V3

1. *Create "and", "or" and "not" gates*

```c
struct Circuit;
```

```c
// Function to create a gate within a circuit
// gate_type : "and", "or",
// Return value is unique id for that gate in circuit
int add_and_gate(Circuit *c);
int add_or_gate(Circuit *c);
int add_not_gate(Circuit *c);
```

# API Design : Connecting Gates

2. *Connect gates together*

```
struct Circuit;

// Struct to represent all gates
struct Gate;

// Function to create a gate within a circuit
Gate *add_gate(Circuit *c, string gate_type);
```

# API Design : Connecting Gates V2a

2. *Connect gates together*

```
struct Circuit;

// Struct to represent all gates
struct Gate;

// Function to create a gate within a circuit
Gate *add_gate(Circuit *c, string gate_type);


// Set input destInputIndex on dest to output of src
void set_gate_input(Circuit *c,
      Gate *dest, int destInputIndex,
      Gate *src
);
```

# API Design : Connecting Gates V2b

*2. Connect gates together*

```
struct Circuit;

// Struct to represent all gates
struct Gate;

// Function to create a gate within a circuit
Gate *add_gate(Circuit *c, string gate_type);


// Add the output of src to the set of inputs for dst
void add_gate_input(Circuit *c, Gate *dst, Gate *src);
```

# API Design : Connecting Gates V1a

2. *Connect gates together*

```
struct Circuit;

// Structs to represent gates
struct AndGate;
struct OrGate;
struct NotGate;
```

# API Design : Connecting Gates V1a

*2. Connect gates together*

```
struct Circuit;

// Structs to represent gates
struct AndGate;
struct OrGate;
struct NotGate;

// Add the output of src to the set of inputs for dst
void add_gate_input(Circuit *c, AndGate *dst, AndGate *src);
void add_gate_input(Circuit *c, AndGate *dst, OrGate *src);
void add_gate_input(Circuit *c, AndGate *dst, NotGate *src)
void add_gate_input(Circuit *c, OrGate *dst,  AndGate *src);
void add_gate_input(Circuit *c, OrGate *dst,  OrGate *src);
void add_gate_input(Circuit *c, OrGate *dst,  NotGate *src);
...
```

# API Design: observations

```
struct AndGate;
struct OrGate;
struct NotGate;

void add_gate_input(Circuit *c, AndGate *dst, AndGate *src);
void add_gate_input(Circuit *c, AndGate *dst, OrGate *src);
void add_gate_input(Circuit *c, AndGate *dst, NotGate *src)
void add_gate_input(Circuit *c, OrGate *dst,  AndGate *src);
void add_gate_input(Circuit *c, OrGate *dst,  OrGate *src);
...
```

There could be may different types of gate, but they are all still a type of “gate”

Can be handled using inheritance

# API Design: observations

```
Circuit *create();

Gate *add_and_gate(Circuit *c, string gate_type);

void add_gate_input(Circuit *c, Gate *dst, Gate *src);

void print(Circuit *c);

void destroy(Circuit *c);
```

There is something special about the circuit pointer.

It represents a single “thing” being:

created, modified, queried, destroyed, ...

# API Design : the case for OOP

Many APIs model “things” with state+compute

(data + functions)

OOP makes it easier to create such APIs

OOP = Object Oriented Programming

# Disclaimer : Objects are *not* magic

*OOP is one paradigm*: it is **not** the most important one
*C++ supports objects*: but it also supports other paradigms
*Recent/hot languages don't use objects*: Rust, Go, ...

A lot of extremely important projects "ban" objects

- *OS*: Linux, Windows, OSX
- *Transport*: cars, planes, submarines...
- *Embedded*: routers, satellites, base-stations, ...

OOP is just one useful tool in the programming tool-box

# An intro to objects

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;
};

void matrix_resize(Matrix *mat, int rows, int cols)
{
    mat->rows=rows;
    mat->cols=cols;
    mat->values.resize(rows*cols);
}

void matrix_write(Matrix *mat, int r, int c, float v)
{
    mat->values[r * mat->cols + c] = v;
}

float matrix_read(Matrix *mat, int r, int c)
{
    return mat->values[r * mat->cols + c];
}
```

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;
};
```

```
void matrix_resize(Matrix *mat, int rows, int cols)
{
    mat->rows=rows;
    mat->cols=cols;
    mat->values.resize(rows*cols);
}

void matrix_write(Matrix *mat, int r, int c, float v)
{
    mat->values[r * mat->cols + c] = v;
}

float matrix_read(Matrix *mat, int r, int c)
{
    return mat->values[r * mat->cols + c];
}
```

# Structs + functions to classes

*Move functions inside the class*

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void matrix_resize(Matrix *mat, int rows, int cols)
    {
        mat->rows=rows;
        mat->cols=cols;
        mat->values.resize(rows*cols);
    }

    void matrix_write(Matrix *mat, int r, int c, float v)
    {
        mat->values[r * mat->cols + c] = v;
    }

    float matrix_read(Matrix *mat, int r, int c)
    {
        return mat->values[r * mat->cols + c];
    }
};
```

# Structs + functions to classes

*Call the pointer to instance "this"*

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void matrix_resize(Matrix *this, int rows, int cols)
    {
        this->rows=rows;
        this->cols=cols;
        this->values.resize(rows*cols);
    }

    void matrix_write(Matrix *this, int r, int c, float v)
    {
        this->values[r * this->cols + c] = v;
    }

    float matrix_read(Matrix *this, int r, int c)
    {
        return this->values[r * this->cols + c];
    }
};
```

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void matrix_resize(Matrix *this, int rows, int cols)
    {
        this->rows=rows;
        this->cols=cols;
        this->values.resize(rows*cols);
    }

    void matrix_write(Matrix *this, int r, int c, float v)
    {
        this->values[r * this->cols + c] = v;
    }

    float matrix_read(Matrix *this, int r, int c)
    {
        return this->values[r * this->cols + c];
    }
};
```

# Structs + functions to classes

*Remove the "this" parameter*

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void matrix_resize(int rows, int cols)
    {
        this->rows=rows;
        this->cols=cols;
        this->values.resize(rows*cols);
    }

    void matrix_write(int r, int c, float v)
    {
        this->values[r * this->cols + c] = v;
    }

    float matrix_read(int r, int c)
    {
        return this->values[r * this->cols + c];
    }
};
```

# Structs + functions to classes

*The class declaration*

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void matrix_resize(int rows, int cols);
    void matrix_write(int r, int c, float v);
    float matrix_read(int r, int c);
};
```

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void matrix_resize(int rows, int cols);
    void matrix_write(int r, int c, float v);
    float matrix_read(int r, int c);
};


int main()
{
    Matrix mat;

    mat.matrix_resize(10,10);

    for(int i=0; i<10; i+=1){
        for(int j=0; j<10; j+=1){
            mat.matrix_write( i, j, sin(i)+cos(j) );
        }
    }
}
```

*Create a local matrix instance*

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void matrix_resize(int rows, int cols);
    void matrix_write(int r, int c, float v);
    float matrix_read(int r, int c);
};


int main()
{
    Matrix mat;

    mat.matrix_resize(10,10);

    for(int i=0; i<10; i+=1){
        for(int j=0; j<10; j+=1){
            mat.matrix_write( i, j, sin(i)+cos(j) );
        }
    }
}
```

*Call functions on the instance*

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void matrix_resize(int rows, int cols);
    void matrix_write(int r, int c, float v);
    float matrix_read(int r, int c);
};


int main()
{
    Matrix mat;

    mat.matrix_resize(10,10);

    for(int i=0; i<10; i+=1){
        for(int j=0; j<10; j+=1){
            mat.matrix_write( i, j, sin(i)+cos(j) );
        }
    }
}
```

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void matrix_resize(int rows, int cols);
    void matrix_write(int r, int c, float v);
    float matrix_read(int r, int c);
};


int main()
{
    Matrix mat;

    mat.matrix_resize(10,10);

    for(int i=0; i<10; i+=1){
        for(int j=0; j<10; j+=1){
            mat.matrix_write( i, j, sin(i)+cos(j) );
        }
    }
}
```

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void resize(int rows, int cols);
    void write(int r, int c, float v);
    float read(int r, int c);
};


int main()
{
    Matrix mat;

    mat.resize(10,10);

    for(int i=0; i<10; i+=1){
        for(int j=0; j<10; j+=1){
            mat.write( i, j, sin(i)+cos(j) );
        }
    }
}
```

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void resize(int rows, int cols);
    void write(int r, int c, float v);
    float read(int r, int c);
};
```

*Can create using new as well*

```
int main()
{
    Matrix *mat = new Matrix;

    mat->resize(10,10);

    for(int i=0; i<10; i+=1){
        for(int j=0; j<10; j+=1){
            mat->write( i, j, sin(i)+cos(j) );
        }
    }
    delete mat;
}
```

# Classes : initial principles

- Classes are structs with methods
  - Move functions *inside* the data-type
  - A function inside a class is called a ***method***
- The `this` pointer is special
  - It always points to the current object instance
  - Only works within a method (function inside a class)
- You call call methods using `.` or `->`
  - The rules are the same as for data on structs
  - You’ve already been doing this: e.g. vectors and cin/cout

# Structs + functions to classes

*Remove the "this" parameter*

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void resize(int rows, int cols)
    {
        this->rows=rows;
        this->cols=cols;
        this->values.resize(rows*cols);
    }

    void write(int r, int c, float v)
    {
        this->values[r * this->cols + c] = v;
    }

    float read(int r, int c)
    {
        return this->values[r * this->cols + c];
    }
};
```

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void resize(int rows, int cols)
    {
        this->rows=rows;
        this->cols=cols;
        this->values.resize(rows*cols);
    }

    void write(int r, int c, float v)
    {
        this->values[r * this->cols + c] = v;
    }

    float read(int r, int c)
    {
        return this->values[r * this->cols + c];
    }
};
```

# Structs + functions to classes

*Member variables are automatically in scope – can omit* `this->`

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void resize(int rows, int cols)
    {
        rows=rows;
        cols=cols;
        values.resize(rows*cols);
    }

    void write(int r, int c, float v)
    {
        values[r * cols + c] = v;
    }

    float read(int r, int c)
    {
        return values[r * cols + c];
    }
};
```

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void resize(int rows, int cols)
    {
        rows=rows;
        cols=cols;
        values.resize(rows*cols);
    }

    void write(int r, int c, float v)
    {
        values[r * cols + c] = v;
    }

    float read(int r, int c)
    {
        return values[r * cols + c];
    }
};
```

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void resize(int rows, int cols)
    {
        rows=rows;
        cols=cols;
        values.resize(rows*cols);
    }

    void write(int r, int c, float v)
    {
        values[r * cols + c] = v;
    }

    float read(int r, int c)
    {
        return values[r * cols + c];
    }
};
```

*Beware shadowing of member variables by parameters*

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void resize(int rows, int cols)
    {
        this->rows=rows;
        this->cols=cols;
        values.resize(rows*cols);
    }

    void write(int r, int c, float v)
    {
        values[r * cols + c] = v;
    }

    float read(int r, int c)
    {
        return values[r * cols + c];
    }
};
```

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void resize(int _rows, int _cols)
    {
        rows=_rows;
        cols=_cols;
        values.resize(rows*cols);
    }

    void write(int r, int c, float v)
    {
        values[r * cols + c] = v;
    }

    float read(int r, int c)
    {
        return values[r * cols + c];
    }
};
```

# Structs + functions to classes

```
struct Matrix
{
    int m_rows;
    int m_cols;
    vector<float> m_values;

    void resize(int rows, int cols)
    {
        m_rows=rows;
        m_cols=cols;
        m_values.resize(m_rows*m_cols);
    }

    void write(int r, int c, float v)
    {
        m_values[r * m_cols + c] = v;
    }

    float read(int r, int c)
    {
        return m_values[r * m_cols + c];
    }
};
```

# Classes : implicit member access

- Class members are always in scope in methods
  - You don't need to specify `this->`
  - Applies to both member variables and methods
- Watch out for aliasing of symbols (names)
  - It's easy to end up with parameters shadowing members
  - Can modify names or use to `this->` disambiguate
- Some people use naming conventions for members
  - e.g. use `m_` prefix on member variables
  - e.g. use `_` suffix on member variables
  - This can be useful, but is a matter of choice

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void resize(int rows, int cols);
    void write(int r, int c, float v);
    float read(int r, int c);
};


int main()
{
    Matrix mat;

    mat.resize(10,10);

    for(int i=0; i<10; i+=1){
        for(int j=0; j<10; j+=1){
            mat.write( i, j, sin(i)+cos(j) );
        }
    }
}
```

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void resize(int rows, int cols);
    void write(int r, int c, float v);
    float read(int r, int c);
};


int main()
{
    Matrix mat;

    mat.resize(10,10);

    for(int i=0; i<10; i+=1){
        for(int j=0; j<10; j+=1){
            mat.write( i, j, sin(i)+cos(j) );
        }
    }
}
```

*Create instance, then initialise*

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void resize(int rows, int cols);
    void write(int r, int c, float v);
    float read(int r, int c);
};


int main()
{
    Matrix mat;

    mat.resize(10,10);

    for(int i=0; i<10; i+=1){
        for(int j=0; j<10; j+=1){
            mat.write( i, j, sin(i)+cos(j) );
        }
    }
}
```

*Create instance, then initialise*

*Is it in a valid state at this point?*

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    void resize(int rows, int cols);
    void write(int r, int c, float v);
    float read(int r, int c);
};


int main()
{
    Matrix mat(10,10);



    for(int i=0; i<10; i+=1){
        for(int j=0; j<10; j+=1){
            mat.write( i, j, sin(i)+cos(j) );
        }
    }
}
```

*Create **and** initialise instance*

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    Matrix(int rows, int cols)
    {
       resize(rows,cols);
    }

    void resize(int rows, int cols);
    void write(int r, int c, float v);
    float read(int r, int c);
};

int main()
{
    Matrix mat(10,10);

    for(int i=0; i<10; i+=1){
        for(int j=0; j<10; j+=1){
            mat.write( i, j, sin(i)+cos(j) );
        }
    }
```

*Calls class* ***constructor***

*Create* ***and*** *initialise instance*

# Classes : constructors

- Classes describe a type with state and computation
  - We want to ensure it is always in a valid state
  - Want to avoid “in-between” states : *created but not valid*
- A **constructor** is used to setup a new object
  - A constructor is a method with same name as the class
  - *Does not have a return type* : it “returns” the instance
  - Can have zero or more parameters
- Classes can also have a ***destructor***
  - Method called when an instance is destroyed
  - We don’t need them yet : important if using `new`/`delete`

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    Matrix(int rows, int cols);

    void resize(int rows, int cols);
    void write(int r, int c, float v);
    float read(int r, int c);
};

int main()
{
    Matrix mat(10,10);

    // Code, code, code

    mat.write( 3, 4, 2.32);
}
```

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    Matrix(int rows, int cols);

    void resize(int rows, int cols);
    void write(int r, int c, float v);
    float read(int r, int c);
};

int main()
{
    Matrix mat(10,10);

    // HACK: matrix needs to be smaller
    mat.values.resize(4*5);

    mat.write( 3, 4, 2.32);
}
```

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    Matrix(int rows, int cols);

    void resize(int rows, int cols);
    void write(int r, int c, float v);
    float read(int r, int c);
};

int main()
{
    Matrix mat(10,10);

    // HACK: matrix needs to be smaller
    mat.values.resize(4*5);

    mat.write( 3, 4, 2.32);
}
```

```
void write(int r, int c, float v)
{
    values[r * cols + c] = v;
}
```

# Structs + functions to classes

```
struct Matrix
{
    int rows;
    int cols;
    vector<float> values;

    Matrix(int rows, int cols);

    void resize(int rows, int cols);
    void write(int r, int c, float v);
    float read(int r, int c);
};

int main()
{
    Matrix mat(10,10);

    // HACK: matrix needs to be smaller
    mat.values.resize(4*5);

    mat.write( 3, 4, 2.32);
}
```

```
void write(int r, int c, float v)
{
  assert(values.size()==rows*cols);
  values[r * cols + c] = v;
}
```

*Class assumptions are broken due to unconstrained changes*

# Structs + functions to classes

```
class Matrix
{
private:
    int rows;
    int cols;
    vector<float> values;

public:
    Matrix(int rows, int cols);

    void resize(int rows, int cols);
    void write(int r, int c, float v);
    float read(int r, int c);
};

int main()
{
    Matrix mat(10,10);

    // HACK: matrix needs to be smaller
    mat.values.resize(4*5);

    mat.write( 3, 4, 2.32);
}
```

# Structs + functions to classes

*A class manages its own state, and protects it from modification*

```
class Matrix
{
private:
    int rows;
    int cols;
    vector<float> values;

public:
    Matrix(int rows, int cols);

    void resize(int rows, int cols);
    void write(int r, int c, float v);
    float read(int r, int c);
};

int main()
{
    Matrix mat(10,10);

    // HACK: matrix needs to be smaller
    mat.values.resize(4*5);

    mat.write( 3, 4, 2.32);
}
```

# Structs + functions to classes

```
class Matrix
{
private:
    int rows;
    int cols;
    vector<float> values;

public:
    Matrix(int rows, int cols);

    void resize(int rows, int cols);
    void write(int r, int c, float v);
    float read(int r, int c);
};

int main()
{
    Matrix mat(10,10);

    // HACK: matrix needs to be smaller
    mat.values.resize(4*5);

    mat.write( 3, 4, 2.32);
}
```

*Private members can only be accessed from inside methods*

# Structs + functions to classes

```
class Matrix
{
private:
    int rows;
    int cols;
    vector<float> values;

public:
    Matrix(int rows, int cols);

    void resize(int rows, int cols);
    void write(int r, int c, float v);
    float read(int r, int c);
};

int main()
{
    Matrix mat(10,10);

    // HACK: matrix needs to be smaller
    mat.values.resize(4*5);

    mat.write( 3, 4, 2.32);
}
```

*Public members can be accessed by anyone*

# Structs + functions to classes

```
class Matrix
{
private:
    int rows;
    int cols;
    vector<float> values;

public:
    Matrix(int rows, int cols);

    void resize(int rows, int cols);
    void write(int r, int c, float v);
    float read(int r, int c);
};

int main()
{
    Matrix mat(10,10);

    // HACK: matrix needs to be smaller
    mat.values.resize(4*5);

    mat.write( 3, 4, 2.32);
}
```

*Compiler error : member variable "values" is inaccessible*

# Structs + functions to classes

```
class Matrix
{
private:
    int rows;
    int cols;
    vector<float> values;

public:
    Matrix(int rows, int cols);

    void resize(int rows, int cols);
    void write(int r, int c, float v);
    float read(int r, int c);
};

int main()
{
    Matrix mat(10,10);

    // matrix needs to be smaller
    mat.resize(4,5);

    mat.write( 3, 4, 2.32);
}
```

# Classes : access modifiers

- Classes can control access to their own members
  - `struct`: all members are ***public*** by default
  - `class`: all members are ***private*** by default
- Class member variables are *usually* kept private
  - The class wants to avoid directly manipulation of data
  - Need to maintain invariants and assumptions about state
- Class methods may be public or private
  - *public* : the API for interacting with and using objects
  - *private* : internal helper functions

# Classes : terminology

**class** : a type which combines data and functions

**member** : named data or function defined in a class

**member variable** (property): named data defined in a class

**method** (member function) : named function define in a class

**object** (instance) : an instance of a class

**this** : pointer to the object a method is running on

**constructor** : method used to initialize new instances of a class

**destructor** : method used to destroy instances of a class

**access modifiers** : used to control access to members of a class

# Objects : they're all around us…

- Object syntax explains some of the things we've seen
  - Constructing strings: `string s("Hello");`
  - Resizing vectors: `vec.resize(10);`
  - Checking if input is one: `cin.fail();`
- There are two main things left to explain `vector<T>`
  - *Templates*: the ability to specify T
  - *Overloading*: adding support for array-like indexing
- We're going to stay with "plain" objects for a while
  - Build on and explore the ideas introduced here
  - Polymorphism and inheritance come after