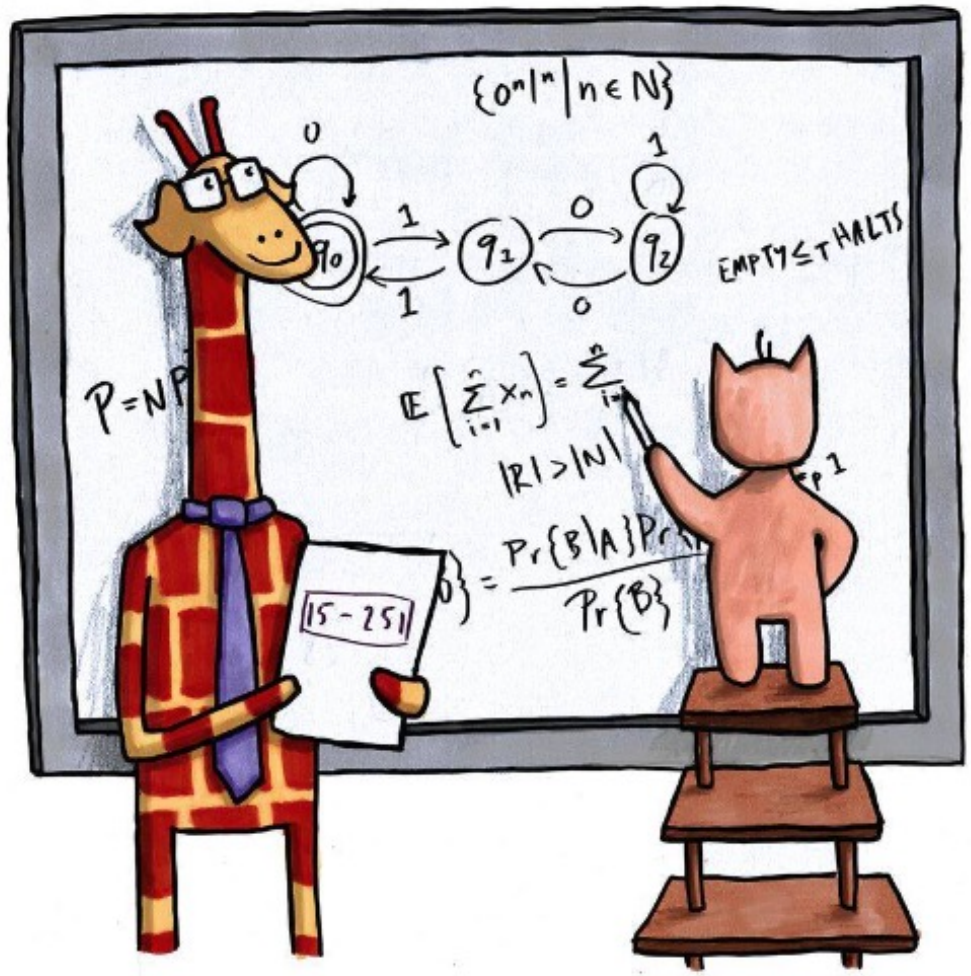
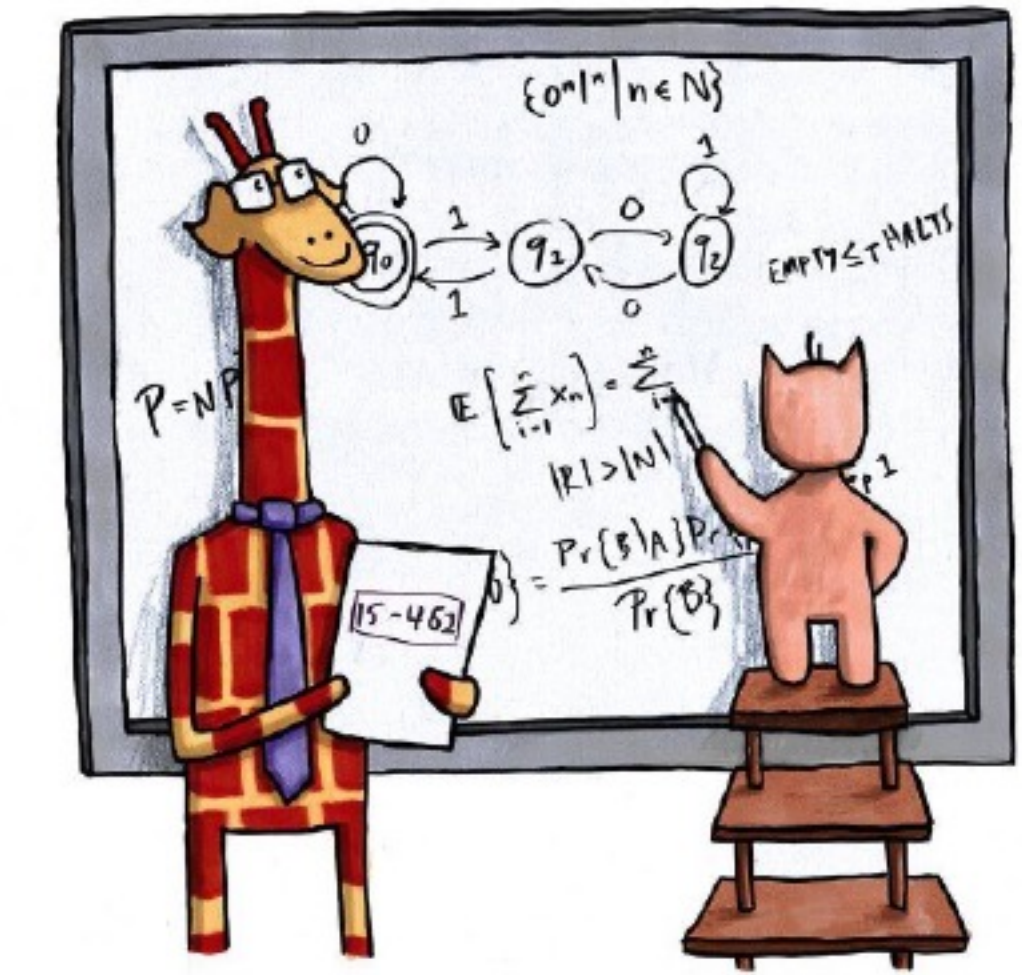
C++: A Programmer's Perspective

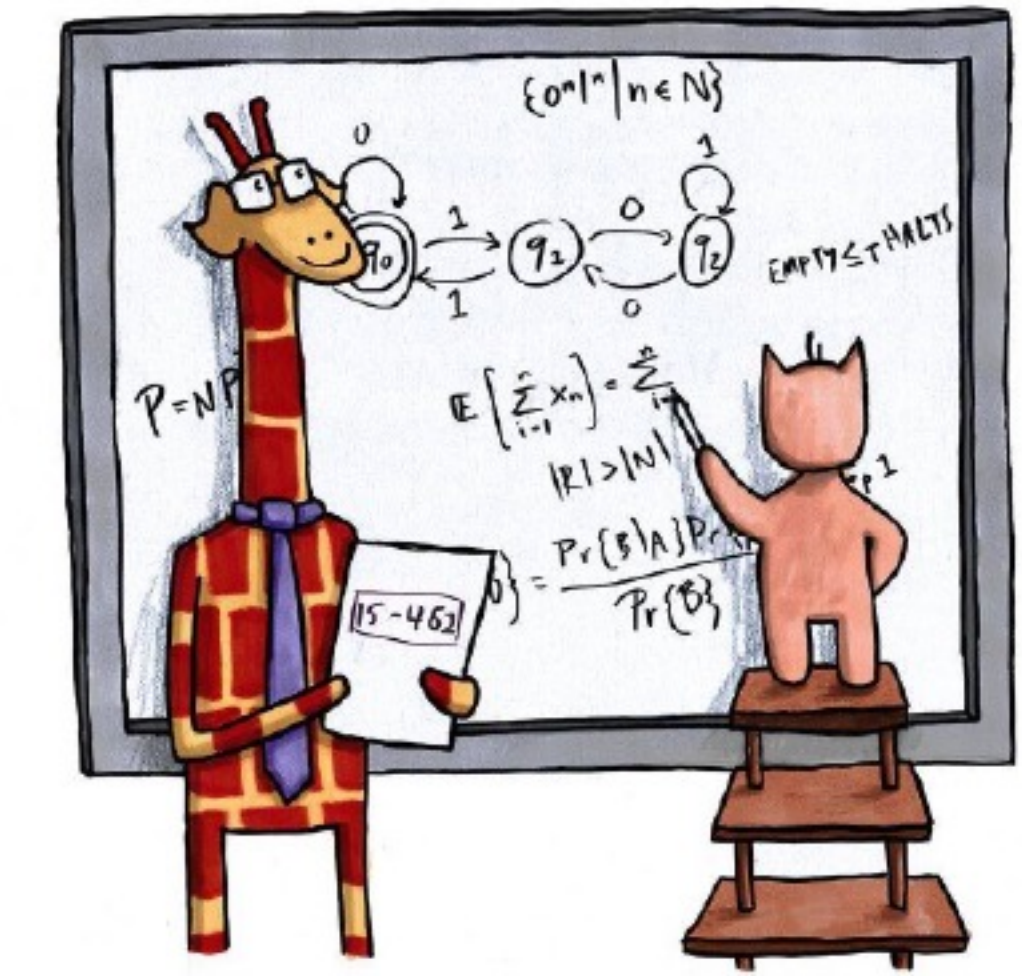
with foreword from max slater



math is hard, but you don't have to do it alone!



math is hard.



math is art.

- Introduction To C++
- C++ Concepts
- Closing Message

C++

- C++ was developed by Bell Labs in 1979 as an extension of the C language
  - Goal was to add object-oriented programming into the C language
  - Still wanted to maintain low-level functionality of C

- Called “C with Classes”
  - Changed to C++ in 1983

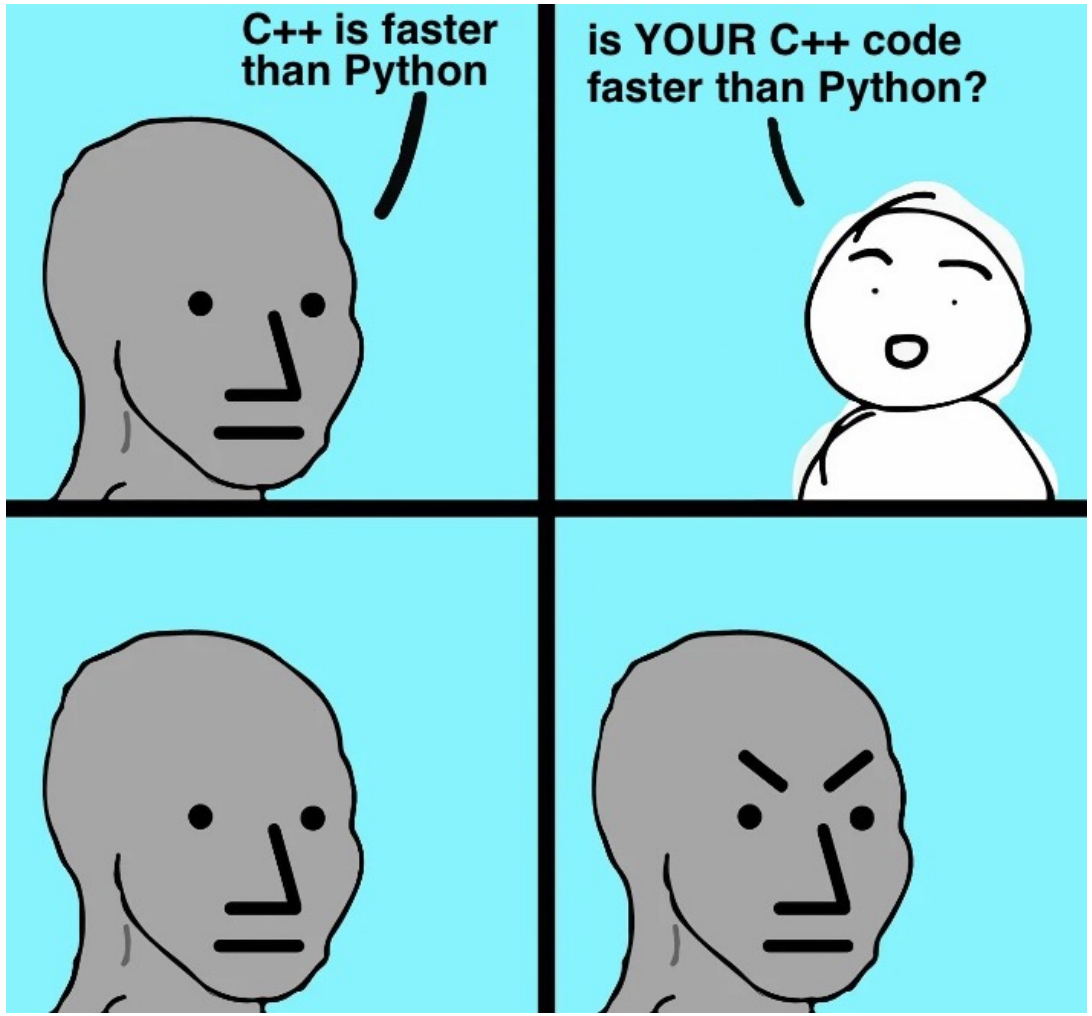
- Every few years a new C++ standard comes out
  - C++11
  - C++14
  - C++17

Why C++ For Graphics

- *“I want a fast language, not a safe language”* -- max slater, probably

- Computer graphics requires quick access of large data
  - Pixel buffers
  - Geometry buffers
  - Textures
  - Offscreen buffers
    - C++ is very efficient at handling large data

- Many graphics APIs already exist in, or work closely with C++
  - OpenGL
  - Vulkan
  - Direct3D



Why C++ For Graphics

- *"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off"* -- Bjarne Stroustrup (1986) creator of C++

- You are responsible for dealing with your own memory
  - Some safety features in C++, but since we're working with large amounts of memory, easy run out of application memory

- With a lot of memory, easy to end up in the wrong place/index

Why C++ For Graphics

- For many, this will be your first time using C++
  - This course is designed to teach you BOTH graphics and C++
- C++ is ubiquitous in graphics academics, research, and industry
  - Having a working knowledge of C++ is essential if continuing graphics
- Another language you can add to your resume :)

WALL-E (2008) Pixar

- ~~Introduction To C++~~
- C++ Concepts
- Closing Message

Classes

- Classes in C++ are defined with a class name and function/variable definitions
- `public:` values are accessible outside the class
- `private:` values are not accessible outside classes and can only be accessed inside class functions
- `protected:` values are private except to other classes that are derived from the class

```
class A {

public:
    int x;
private:
    int y;
protected:
    int z:

};

class B: public A {
    void test1() {
        // works: B has access to public and protected
        return x + z;
    }
    void test2() {
        // fails: B doesn't have access to private
        return y;
    }
}
```

Classes

- `test1()` can execute because public and protected values of the base class A are visible to the derived class B
- `test2()` fails because a derived class does not have access to private values of a base class.
  - To fix this, we can declare B as a friend class of A
- Classes trust friends with their private values!

```
class A {
public:
    int x;
private:
    int y;
    // specify B is a friend class of A
    friend class B
protected:
    int z:
};
class B: public A {
    void test1() {
        // works: B has access to public and protected
        return x + z;
    }
    void test2() {
        // works: B is a friend class and
        // can access private A values
        return y;
    }
}
```

Classes

- In Scotty3D, our classes obey these access patterns.
  - Here we have the `Vertex` class.
- The `Vertex` constructor is a private field
  - In Scotty3D, we do not want just anyone creating `Vertex` objects
  - We declare the constructor as private and give only certain classes access to create `Vertex` objects using the `friend class` specifier

```
class Vertex {
public:

    HalfedgeRef& halfedge() {return _halfedge;}
    HalfedgeCRef halfedge() const {return _halfedge;}

    bool on_boundary() const;
    unsigned int degree() const;
    Vec3 normal() const;
    Vec3 center() const;
    Vec3 neighborhood_center() const;
    unsigned int id() const {return _id;}

    Vec3 pos;

private:
    Vertex(unsigned int id) : _id(id) {}
    Vec3 new_pos;
    bool is_new = false;
    unsigned int _id = 0;
    HalfedgeRef _halfedge;
    friend class Halfedge_Mesh;
};
```

Classes

- Some features we can edit directly without using a class function, but other features are `private` and cannot be edited
- So how do we edit `new_pos` ?

```
Vertex *c = vertexList[i];

// works: public variable
c->pos = Vec3(1.f,0.f,0.f);

// fails: private variable
c->new_pos = Vec3(1.f,0.f,0.f);
```

```
class Vertex {
public:

    HalfedgeRef& halfedge() {return _halfedge;}
    HalfedgeCRef halfedge() const {return _halfedge;}

    bool on_boundary() const;
    unsigned int degree() const;
    Vec3 normal() const;
    Vec3 center() const;
    Vec3 neighborhood_center() const;
    unsigned int id() const {return _id;}

    Vec3 pos;

private:
    Vertex(unsigned int id) : _id(id) {}
    Vec3 new_pos;
    bool is_new = false;
    unsigned int _id = 0;
    HalfedgeRef _halfedge;
    friend class Halfedge_Mesh;
};
```

Classes

- Classes can inherit attributes from other classes, gaining access to public and private values in those classes

- The `Halfedge_Mesh` class is built using `Vertex`, Edge, Face, and `Halfedge` components
  - We can declare `Halfedge_Mesh` as a `friend class` in the `Vertex` class

```
class Vertex {

...

private:

    ...
    // we give Halfedge_Mesh access to private values
    friend class Halfedge_Mesh;
};

class Halfedge_Mesh {
public:
    // access to public & private values in Vertex class
    class Vertex;
    class Edge;
    class Face;
    class Halfedge;

    std::optional<FaceRef> bevel_vertex(VertexRef v);
    std::optional<FaceRef> bevel_edge(EdgeRef e);
    std::optional<FaceRef> bevel_face(FaceRef f);
}
```

Classes

- Now in our `Halfedge_Mesh` class, we can write functions that access these private values from the `Vertex` class
- We chose to make variables in classes `private` in order to ensure classes that we don't want modifying these variables will never have access to it

```
class Vertex {

...

private:

    ...
    // we give Halfedge_Mesh access to private values
    friend class Halfedge_Mesh;
};

class Halfedge_Mesh {
public:
    // access to public & private values in Vertex class
    class Vertex;
    class Edge;
    class Face;
    class Halfedge;

    std::optional<FaceRef> bevel_vertex(VertexRef v);
    std::optional<FaceRef> bevel_edge(EdgeRef e);
    std::optional<FaceRef> bevel_face(FaceRef f);
}
```

```
auto halfedge_Mesh::bevel(VertexRef v) {

    // works: public variable
    v->pos = Vec3(1.f,0.f,0.f);
    // works: Halfedge_Mesh is friend
    v->new_pos = Vec3(1.f,0.f,0.f);
}
```

Templates

```
class Cache {

public:

    Cache(int N) {
        data = calloc(N, sizeof(datatype));
        freq = calloc(N, sizeof(int));
        size = N;
    }
    datatype get(int idx) {
        freq[idx]++;
        return data[idx];
    }

private:

    datatype *data;
    int *freq;
    int size;
}
```

- Templates help define a generic `datatype` that you can use to write versatile code without explicitly defining the interface for each type
  - Your compiler writes the template code only if it is called, otherwise the compiler avoids it
  - Templating can be thought of as getting the compiler to write your code

- Say we'd like to create a new set of classes that will track query usage per element to help for analytics. We can call this class our `cache`

Templates

```
class CacheInt {

public:

    CacheInt(int N) {
        data = calloc(N, sizeof(int));
        freq = calloc(N, sizeof(int));
        size = N;
    }
    int get(int idx) {
        freq[idx]++;
        return data[idx];
    }

private:

    int *data;
    int *freq;
    int size;
}
```

- Here is our class for `ints`

Templates

```
class CacheFloat {

public:

    CacheFloat (int N) {
        data = calloc(N, sizeof(float));
        freq = calloc(N, sizeof(float));
        size = N;
    }
    float get(int idx) {
        freq[idx]++;
        return data[idx];
    }

private:

    float *data;
    int *freq;
    int size;
}
```

- Here is our class for `floats`

Templates

```
class CacheDouble {

public:

    CacheDouble (int N) {
        data = calloc(N, sizeof(double));
        freq = calloc(N, sizeof(double));
        size = N;
    }
    double get(int idx) {
        freq[idx]++;
        return data[idx];
    }

private:

    double *data;
    int *freq;
    int size;
}
```

- Here is our class for `doubles`
  - This is getting to be a lot of code…

Templates

```
Template<typename T>
class Cache {

public:

    Cache(int N) {
        data = calloc(N, sizeof(T));
        freq = calloc(N, sizeof(int));
        size = N;
    }
    T get(int idx) {
        freq[idx]++;
        return data[idx];
    }

private:

    T *data;
    int *freq;
    int size;
}
```

- We can use `Template<typename T>` to create a generic datatype and bind it on compile time
- Then we can create instances of `int`, `float` and `double` caches easily

```
// creates int class
Cache<int> a = Cache<int>(10);
// creates float class
Cache<float> b = Cache<float>(10);
// creates double class
Cache<double> c = Cache<double>(10);
```

Templates

```
Template<typename T>
class Cache {

public:

    Cache(int N) {
        data = calloc(N, sizeof(T));
        freq = calloc(N, sizeof(int));
        size = N;
    }
    T get(int idx) {
        freq[idx]++;
        return data[idx];
    }

private:

    T *data;
    int *freq;
    int size;
}
```

- The best way to think about this is that every time we create a new template datatype for the `Cache` class, it makes a copy of the string defining the class and swaps out every instance of the `typename T` before inserting it into the file and compiling it
  - If we never make a call to `Cache` with `type T`, then the template is ignored and we never compile it (yet another optimization on the compiler's part!)

```
// creates int class
Cache<int> a = Cache<int>(10);
// creates float class
Cache<float> b = Cache<float>(10);
// double class never created!
```

Templates

- We use templates a lot in Scotty3D, and you'll even work to implement some template classes in the Animation unit. For example:

```
Template<typename T>
class Spline {
public:

    T at(float time) const;

    void set(float time, T value) { control_points[time] = value; }
    void erase(float time) { control_points.erase(time); }
    bool has(float t) const { return control_points.count(t); }
    bool any() const { return !control_points.empty(); }
    void clear() { control_points.clear(); }

private:
    // records keyframe values at specific times
    std::map&ltfloat, T&gt control_points;
};
```

Templates

```
Spline<int> S;
S.add(0.f, 0);
S.add(1.f, 61848);
// prints 15462
printf("%d\n", S.at(0.25))
```

- The program will then the below string of code, replacing `typename T` with `Spline`, making a strong-typed class interface for the type

```
Template<typename T>
class Spline {
public:

    T at(float time) const;

    void set(float time, T value) { control_points[time] = value; }
    void erase(float time) { control_points.erase(time); }
    bool has(float t) const { return control_points.count(t); }
    bool any() const { return !control_points.empty(); }
    void clear() { control_points.clear(); }

private:
    // records keyframe values at specific times
    std::map&ltfloat, T&gt control_points;
};
```

Templates

- Many other library classes and data structures also use templates to exploit generic types. C++ classes like the ones below use templates to avoid redundant coding:

```
std::map<T1, T2> A;              // list of key-value pairs
std::unordered_map<T1, T2> B;    // unordered list of key-value pairs
std::set<T> C;                   // list without duplicates
std::unordered_set<T> D;         // unordered list without duplicates
std::list<T> E;                  // dumb list
std::vector<T> F;                // fancy list
std::deque<T> G;                 // list with multi-side insertion/deletion
std::pair<T1, T2> H;             // two-element list
```

Iterators

- Iterators allow you it iterate through ordered and unordered structs (like `sets` and `unordered maps`) using the `::iterator` attribute

- We can see that `mesh.vertices` holds a list of `Vertex` objects. `vertices.begin()` returns the first iterator in our list, and we increment that until we reach the last iterator `vertices.end()` (non-inclusive)
  - We typedef `list<Vertex>::iterator` as `VertexIter` for easy notation

```
class HalfedgeMesh {

public:

    list<Vertex> vertices;
    ...
    typedef list<Vertex>::iterator VertexIter;
    ...
    VertexIter verticesBegin() { return vertices.begin(); }
    VertexIter verticesEnd() { return vertices.end(); }
    ...

};
```

Iterators

- Our normal approach in C would be to determine the number of vertices in the list and use a for loop with indexing to query for each vertex and update its properties
  - We can instead use iterators in our for-loop to simplify the code and speed up the process

```
class HalfedgeMesh {

public:

    list<Vertex> vertices;
    ...
    typedef list<Vertex>::iterator VertexIter;
    ...
    VertexIter verticesBegin() { return vertices.begin(); }
    VertexIter verticesEnd() { return vertices.end(); }
    ...

};
```

```
HalfedgeMesh& mesh;

// iterate over list<Vertex> elements
for(VertexIter v = mesh.verticesBegin(); v != mesh.verticesEnd(); v++) {
    v->position = v->newPosition;
}
```

Iterators

- An easy way to think about iterators are as pointers. We can iterate through our vector of ints and dereference the iterator we are on to get the value it points to. By incrementing our iterator, we are jumping `sizeof(datatype)` bytes in memory to get a pointer to the next value

```
vector<int> a = { 1, 5, 4, 6, 2 };
printf("I love ");

for(vector<int>::iterator ptr = a.begin(); ptr != a.end(); ptr++) {
    printf("%d", *ptr);
}
```

Auto

- Sometimes (especially in graphics) datatypes in C++ can be very long. A good programmer will recognize when a dataype should be obvious and instead will alias the original datatype using the `auto` keyword. Consider the following example:

```
const std::vector<Mesh::Index>& Mesh::indices() const {
    return _idxs;
}

// long, annoying, hard to type
Mesh& mesh;
const std::vector<Mesh::Index>& idx = mesh.indices();
```

Auto

- Sometimes (especially in graphics) datatypes in C++ can be very long. A good programmer will recognize when a dataype should be obvious and instead will alias the original datatype using the `auto` keyword. Consider the following example:

```
const std::vector<Mesh::Index>& Mesh::indices() const {
    return _idxs;
}

// long, annoying, hard to type
Mesh& mesh;
const std::vector<Mesh::Index>& idx = mesh.indices();
```

```
// easy, breezy, beautiful
Mesh& mesh;
const auto& idx = mesh.indices();
```

Const

- `const` prevents us from modifying the state of an object or variable. We can define a variable as const such that we cannot update it, or we can define a function in a class as const such that we cannot modify any of the properties of that class
  - This helps optimize our code (telling the compiler that we don't need write access to the variable) and also helps create safer code by ensuring class variables won't be overwritten

- The following function starts by reading the `_halfedge` of the `Face` class and iterates through each halfedge to get the degree of the face. During the process we only read from but do not write to any class values

```
unsigned int Halfedge_Mesh::Face::degree() const {

    unsigned int d = 0;
    HalfedgeCRef h = _halfedge;

    do {
        d++;
        h = h->next();
    } while (h != _halfedge);

    return d;

}
```

Const

- But wait! what's `HalfedgeCRef`??
- It's a `const_iterator`! But wait! how are we able to update `h` if it's a `const_iterator`?
  - Let's look at a quick example

```
unsigned int Halfedge_Mesh::Face::degree() const {

    unsigned int d = 0;
    HalfedgeCRef h = _halfedge;

    do {
        d++;
        h = h->next();
    } while (h != _halfedge);

    return d;

}
```

```
using HalfedgeCRef = list<Halfedge>::const_iterator;
```

Const

```
int valA = 15462;
int valB = 15418;

int *a = &valA; // normal int pointer
a = &valB;      // success! we can modify what a points too
*a = 15213;     // success! we can modify the value at the address a points to

const int *b = &valA; // pointer to const int
b = &valB;            // success! we can modify what b points too
*b = 15213;           // fails: can't change value of const int

int const *c = &valA; // const pointer to int
c = &valB;            // fails: can't change pointer
*c = 15213;           // success! we can modify the value at the address c points to

const int const *d = &valA; // const pointer to a const int
d = &valB;                  // fails: can't change pointer
*d = 15213;                 // fails: can't change value of const int
```

- We normally define an int pointer as `int*` and can put the `const` symbol before the `int` datatype (`const int *ptr`) or after it (`int const *ptr`). Yet these lead to different behaviors as seen above

Const

- Where we place the `const` keyword matters as to whether we can change the address of the pointer or the value at the location of the pointer.

- With `HalfedgeCRef`, since it is a `const_iterator`, that means that it is an iterator to a const value in the list. Thus, we can change the address that `h` points to, and we never end up changing the actual values in the halfedge list
  - This prevents us from accidentally changing values in the halfedge lists, giving us read-only access to the data in order to compute the degree of the face via halfedge traversal

```
unsigned int Halfedge_Mesh::Face::degree() const {

    unsigned int d = 0;
    HalfedgeCRef h = _halfedge;

    do {
        d++;
        h = h->next();
    } while (h != _halfedge);

    return d;

}
```

```
using HalfedgeCRef = list<Halfedge>::const_iterator;
```

Static

```
struct Mat4 {

    static const Mat4 I;
    static const Mat4 Zero;

    static Mat4 transpose(const Mat4& m);
    static Mat4 inverse(const Mat4& m);
    static Mat4 translate(Vec3 t);
    static Mat4 rotate(float t, Vec3 axis);
    static Mat4 euler(Vec3 angles);
    static Mat4 rotate_to(Vec3 dir);
    static Mat4 rotate_z_to(Vec3 dir);
    static Mat4 scale(Vec3 s);
    static Mat4 axes(Vec3 x, Vec3 y, Vec3 z);
    ...
}
```

```
Vec3 t = Vec3(1.f,2.f,3.f);

// calling a Mat4 static function
// without using an instance to call it
Mat4 m = Mat4::axes(t,t,t);
```

- The `static` keyword can be used in different contexts to mean different things. In general, we use `static` to help define that a resource is shared/accessible in a larger scope than it actually is
  - Inside of a class, we can define both functions and variables as `static`
- For `static` functions, we can call them without creating an instance of the class by reference the `Mat4::` class-name before calling the class function.

Static

- `static` variables are shared among all copies of object instances. In this case, regardless of how many `Mat4` objects we create, they will always share the same `I` (identity) and `Zero` (zero matrix) values

- A program can also define the static variables and call functions without ever needing to create an instance. It does so by defining `I` and `Zero` as globally accessible parameters under the `Mat4` struct namespace

- Once we define a static value, we cannot redefine it, thus making it constant

```
const inline Mat4 Mat4::I = Mat4{Vec4{1.0f, 0.0f, 0.0f, 0.0f},
                Vec4{0.0f, 1.0f, 0.0f, 0.0f},
                Vec4{0.0f, 0.0f, 1.0f, 0.0f},
                Vec4{0.0f, 0.0f, 0.0f, 1.0f}};

const inline Mat4 Mat4::Zero = Mat4{Vec4{0.0f, 0.0f, 0.0f, 0.0f},
                Vec4{0.0f, 0.0f, 0.0f, 0.0f},
                Vec4{0.0f, 0.0f, 0.0f, 0.0f},
                Vec4{0.0f, 0.0f, 0.0f, 0.0f}};
```

Static

```
------------ main.cpp ------------

#include "library.h"
int numCores = 8;
```

```
------------ library.h ------------

int numCores = 1;
```

```
------------ main.cpp ------------

#include "library.h"
int numCores = 8;
```

```
------------ library.h ------------

static int numCores = 1;
```

- Another instance of `static` variables (although not as commonly used) is declaring static global variables. The effect of this is to limit the scope of the variable to the current file only
- Your compiler will give you a warning when linking `library.h` that `numCores` has been redefined. To resolve this issue, we can use the `static` keyword to bind the variables to a file-only scope

Namespace

```
namespace PT {

    class Tri_Mesh {

    public:
        Tri_Mesh() = default;
        Tri_Mesh(const GL::Mesh& mesh);

        BBox bbox() const;
        Trace hit(const Ray& ray) const;

        ...

        void build(const GL::Mesh& mesh);

    private:
        std::vector&ltTri_Mesh_Vert&gt verts;
        BVH&ltTriangle&gt triangles;

    };

}
```

- `Namespace` help us encapsulate data and types into a group for easier referencing. It can also help us avoid conflicts if we have variables and functions of the same name.
- In Scotty3D we use `namespace` to group together different pieces of the graphics pipeline. For example, one namespace you will be seeing a lot in A3 is `PT` (PathTracer).

Namespace

```
namespace PT {

    class Tri_Mesh {

    public:
        Tri_Mesh() = default;
        Tri_Mesh(const GL::Mesh& mesh);

        BBox bbox() const;
        Trace hit(const Ray& ray) const;

        ...

        void build(const GL::Mesh& mesh);

    private:
        std::vector&ltTri_Mesh_Vert&gt verts;
        BVH&ltTriangle&gt triangles;

    };

}
```

- Here, we've enclosed the `Tri_Mesh` class in the namespace `PT`
- If we'd like to reference an instance of this class, we can do so using the `PT::` format

```
class Rig {
...
private:
    ...
    // Tri_Mesh defined in PT namespace
    PT::Tri_Mesh mesh_bvh;
};
```

Namespace

```
namespace PT {

    class Tri_Mesh {

    public:
        Tri_Mesh() = default;
        Tri_Mesh(const GL::Mesh& mesh);

        BBox bbox() const;
        Trace hit(const Ray& ray) const;

        ...

        void build(const GL::Mesh& mesh);

    private:
        std::vector&ltTri_Mesh_Vert&gt verts;
        BVH&ltTriangle&gt triangles;

    };

}
```

- We can also un-enclose namespaces if the name is too long and/or we know we will be using objects from that namespace a lot in a given file
- This can be done with the `using` keyword

```
using namespace PT;
class Rig {
...
private:
    ...
    // access Tri_Mesh without PT
    Tri_Mesh mesh_bvh;
};
```

Virtual

```
class A {
    virtual void undo() {...};
    void redo() {...};
};
class B : public A {
    void undo() {...};
    void redo() {...};
};
```

```
B b;
A *a = &b;
a->undo() // which undo do I call?
a->redo() // which redo do I call?
```

- `virtual` functions allow the program to call a function from a derived class using a base-class pointer of a derived class object
  - Does that not make sense?
  - Good. It shouldn't have
- Say I have two classes `A` and `B`. `B` is publicly derived from `A`, alongside sharing some similar function names

Virtual

```
class A {
    // virtual tells us to go to the derived class
    virtual void undo() {...};
    // program doesn't know of another existence
    // runs this instance
    void redo() {...};
};
class B : public A {
    // program runs this version
    void undo() {...};
    // program doesn't run this version
    void redo() {...};
};
```

```
B b;
A *a = &b;
a->undo() // calls derived instance
a->redo() // calls base instance
```

- We know that `a->undo()` calls the derived instance and `a->redo()` calls the base instance
- Defining virtual functions is a handy tool for runtime polymorphism where we want to override a base function definition in a derived class
  - We can also define virtual functions in base classes as templates for derived classes

Virtual

```
class Action_Base {
    virtual void undo() = 0;
    virtual void redo() = 0;
    friend class Undo;
public:
    virtual ~Action_Base() = default;
};
```

```
class MeshOp : public Action_Base {
    void undo() {
        Scene_Object& obj = scene.get_obj(id);
        obj.set_mesh(mesh);
    }
    void redo() {
        Scene_Object& obj = scene.get_obj(id);
        auto sel = obj.set_mesh(mesh, eid);
        op(obj.get_mesh(), sel);
}
```

- In Scotty3D, we use `virtual` functions in the `Action_Base` class to give us a template for our `MeshOp` class.

Datatypes

- C++ comes with many handy datatypes used for storing and parsing data. Each datatype is implemented in a different fashion, and choosing the correct datatype depends on the task (what kinds of data access patterns you will have).

- The `vector` class is one of the most common classes in C++ meant to handle random-access iterators well.
  - `vector` elements are always held in order, meaning that insertions at the end of the list is `O(1)` but insertions in the middle of the list are `O(n)` since we need to shift elements over. The amortized insertion cost is `O(1)`

```
// continuous memory layout
std::vector<T> v;
// doubly-linked list
std::list<T> v;
```

Datatypes

- An application of the `vector` class includes:

```
// continuous memory layout
std::vector<T> v;
// doubly-linked list
std::list<T> v;
```

```
#include <vector>

Vector<int> a(50);                    // create int vector of size 50
Vector<int> b(10, 15462);             // create int vector of size 10 initialized as 15462
a.size();                             // returns 50
a.push_back(15462);                   // adds 15462 to back of list
a.pop_back();                         // removes last element
Vector<int>::iterator p = a.begin()   // get iterator to first element
while(p < a.end()) {                  // get iterator to last element
(void)a[p];                           // supports random access patterns well
p += randint(0,5); }
```

Memory Management

```
class Joint {
public:
    Joint(unsigned int id) : _id(id) {}
    Joint(unsigned int id, Joint* parent, Vec3 extent) ...
    ~Joint() { for(Joint* j : children) delete j; }
    ...
private:
    std::unordered_set<Joint> children;
    ...
    friend class Skeleton;
    friend class Scene;
};
```

- In C++, you'll need to manage your own memory. That means you'll need to allocate memory when needed and free memory when done
- As an example, the `Joint` class stores an `unordered_set<Joint>` of all the joint's children. When a `Joint` is deleted, the deconstructor `~Joint()` iterates through each child joint and deletes it (think of it as burning the bridge in a linked-list structure)

Memory Management

```
class Joint {
public:
    Joint(unsigned int id) : _id(id) {}
    Joint(unsigned int id, Joint* parent, Vec3 extent) ...
    ~Joint() { for(Joint* j : children) delete j; }
    ...
private:
    std::unordered_set<Joint> children;
    ...
    friend class Skeleton;
    friend class Scene;
};
```

- When we go to add a child joint to our skeleton, we call the `Joint(unsigned int id, Joint* parent, Vec3 extent)` constructor using the `new` keyword.

```
Joint* Skeleton::add_child(Joint* j, Vec3 e) {
    Joint* c = new Joint(next_id++, j, e);    // use the new keyword when allocating
    for(float f : keys()) {
        c->anim.set(f, Quat{});
    }
    j->children.insert(c);
    return c;
}
```

Memory Management

```
class Joint {
public:
    Joint(unsigned int id) : _id(id) {}
    Joint(unsigned int id, Joint* parent, Vec3 extent) ...
    ~Joint() { for(Joint* j : children) delete j; }
    ...
private:
    std::unordered_set<Joint> children;
    ...
    friend class Skeleton;
    friend class Scene;
};
```

```
void Skeleton::erase(Joint* j) {
    if(j->parent) {
        j->parent->children.erase(j);
    } else {
        roots.erase(j);
    }
    erased.insert(j);
}
```

- When we want to delete our joint in our implementation, we can add the joint to an `erased` list

Memory Management

```
class Joint {
public:
    Joint(unsigned int id) : _id(id) {}
    Joint(unsigned int id, Joint* parent, Vec3 extent) ...
    ~Joint() { for(Joint* j : children) delete j; }
    ...
private:
    std::unordered_set<Joint> children;
    ...
    friend class Skeleton;
    friend class Scene;
};
```

- Then when we call `delete` on our `Skeleton` object, we iterate over the joints from both the `roots` and `erased` lists and delete the joints
- Our deconstructor deletes both the reference to the `Skeleton` object and any other objects that it held

```
void Skeleton::erase(Joint* j) {
    if(j->parent) {
        j->parent->children.erase(j);
    } else {
        roots.erase(j);
    }
    erased.insert(j);
}
```

```
Skeleton::~Skeleton() {

    for(Joint* j : roots)
        delete j;

    for(Joint* j : erased)
        delete j;
}
```

References

- **Pass-by-value** creates a copy of the variable so that any modifications to the variable are binded to a local-scope of that function alone

- **Pass-by-reference** gets the address of the variable in memory, allowing the function to modify the variable such that the modification will be present even after the function returns

```
bool hit(Line line, Vec3& pt) const {
    Vec3 n = p.xyz();
    float d = dot(line.dir, n);
    float t = (p.w - dot(line.point, n)) / d
    // can assign to pt
    // value persists after function returns
    pt = line.at(t);
    return t >= 0.0f;
}
```

- In `hit()` we pass in `line` as a value and `pt` as a reference
  - `line` is a copy, and so any modifications made to it will be local in scope to the function
  - `pt` is passed as reference, and any modifications made to it will be saved
  - This is useful when we not only want to return true if our line hits the plane, but we would also want to store the resulting hit location in `pt`
    - Using this layout, we can get multiple pieces of information using references

References

- We can declare functions that automatically cast variables to ref on input and return
- This does not mean that the variable accepting the return must also be a reference. Consider how `h` is being assigned a reference even though it is not a reference type itself
- So then why bother returning a ref?
  - Sometimes we just want reassurance that we're always returning a reference to the same vertex halfedge in our call to `vert->halfedge()`, and that no duplicates are being created

```
class Vertex {
public:
    HalfedgeRef& halfedge() {return _halfedge;}
    ...
private:
    ...
    HalfedgeRef _halfedge;
};
```

```
float totalArea = 0.0f;
// HalfedgeRef type accepts HalfedgeRef&
HalfEdgeRef h = vert->halfedge(); do {
  if(!h->face()->is_boundary()) {
    totalArea += h->face()->area();
  }
  // because it isn't a ref, we can keep updating it
  h = h->twin()->next();
}
while(h != vert->halfedge());
```

~~• Introduction To C++~~

~~• C++ Concepts~~

• Closing Message

Good Luck!

- Did anyone ever tell you what `OpenGL` stands for?
  - Open Graphics Library?
  - Open GPU Lists?
  - Open Generative Language?
- None of the above! `OpenGL` stands for Open Good Luck! That's because despite how hard graphics can be, and how frustrated you might get, you're never alone. We're all here to support you in the graphics community. So take a chance, have fun, and get your hands dirty with some graphics code
- We can't wait to see the things you'll go on to create :)
  -- 15462 Staff