C++: A Programmer's Perspective

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

C++

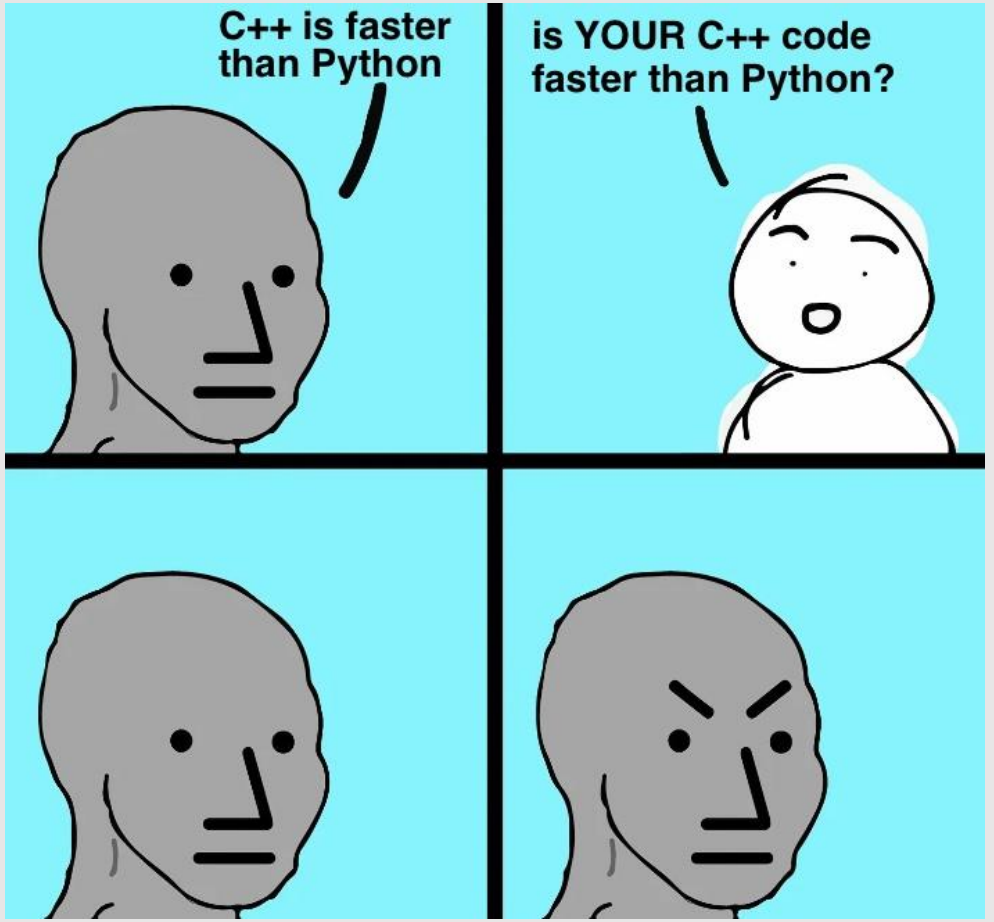
- C++ was developed by Bell Labs in 1979 as an extension of the C language
  - Add object-oriented programming to C
  - Still maintain low-level C functionality
- Called “C with Classes”
  - Changed to C++ in 1983
- Every few years a new C++ standard comes out, breaking project compilations
  - C++11
  - C++14
  - C++17
  - C++20
  - C++23

Why C++ For Graphics

- *"I want a fast language, not a safe language"* -- max slater, former TA

- Computer graphics requires quick access of large data
  - Pixel buffers
  - Geometry buffers
  - Textures
  - Offscreen buffers

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

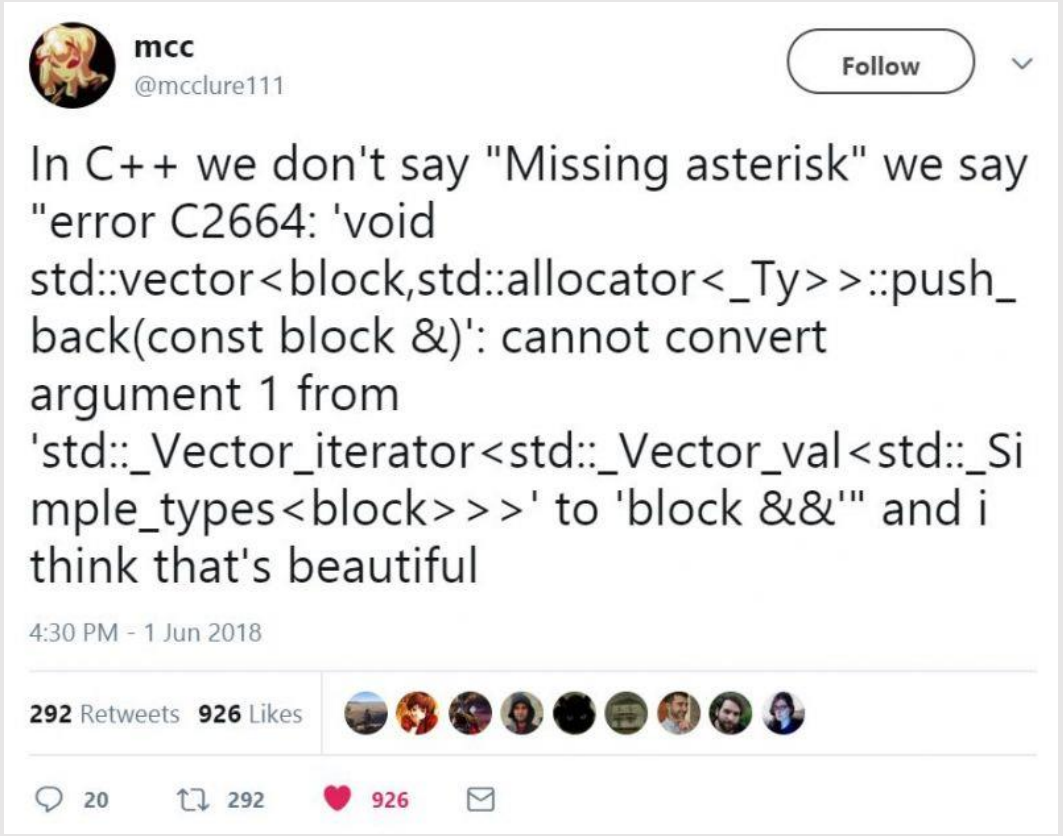


Why C++ For Graphics

- *"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++

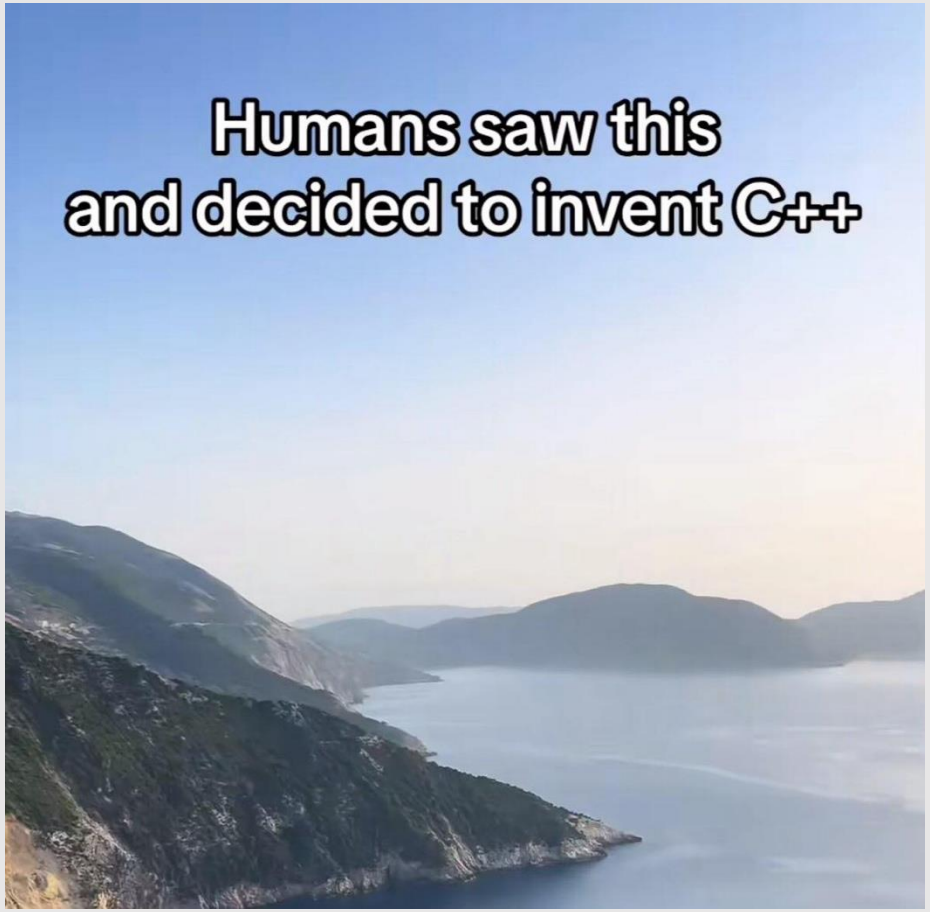
- You are responsible for dealing with your own memory
  - Some safety features in C++, but easy to run out of application memory

- 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 use of C++
  - I'm sorry
- 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 :)



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

Classes

- Classes in C++ are defined with a class name and function/variable definitions
- `public:` values accessible outside the class
- `private:` values not accessible outside the class. Only accessible inside class functions
- `protected:` values private except to other classes 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()` passes because public and protected values of base class A are visible to derived class B
- `test2()` fails because a derived class does not have access to private values of a base class
  - To fix this, 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, the `Vertex` constructor is a `private` field
  - We do not want anyone creating `Vertex` objects
- Constructor is `private`. Only certain classes can 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

- We can edit `public` properties without a class function, but other features are `private` and cannot be edited
- 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

- We chose to make variables in classes `private` to ensure unwanted access is not granted. Access is granted via friend classes
- The `Halfedge_Mesh` class can now edit `new_pos`

```
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 define a generic `datatype` that you can use to write versatile code without defining the type
- **Ex:** We want to create a new set of classes that will track query usage per element for analytics. We 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 `int`

Templates

```
class CacheFloat {

public:

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

private:

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

- Here is our class for `float`

Templates

```
class CacheDouble {

public:

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

private:

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

- Here is our class for `double`
  - 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;
}
```

- Use `Template<typename T>` to create a generic datatype and bind it on compile time
- Then 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;
}
```

- 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

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

Templates

Templates used a lot in Scotty3D, particularly 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, 61448);
// prints 15362
printf("%d\n", S.at(0.25))
```

- The program will copy the below string of code, replacing `typename T` with `int`

```
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

C++ also uses templates in some of its most common data structures:

```
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

- `mesh.vertices` holds a list of `Vertex` objects. `vertices.begin()` returns the first iterator in the list, and we increment that until we reach the last iterator `vertices.end()`
  - `vertices.end()` does not contain a value

```
class HalfedgeMesh {

public:

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

};
```

- We typedef `list<Vertex>::iterator` as `VertexIter` for easy notation

Iterators

- We can use iterators in our for-loop instead of using indices

```
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

Iterators are like pointers. You dereference it to get the value. By incrementing the iterator, we are jumping memory locations to get a pointer to the next value

```
vector<int> a = { 1, 5, 3, 6, 2 };
printf("I love ");

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

Iterators

```
vector<int> a = { 1, 5, 3, 6, 2 };
printf("I love ");

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

Prints: I love 15362

```
vector<int> a = { 1, 5, 3, 6, 2 };
printf("I love ");

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

Prints: I love 153 (or some other incomplete string)

Auto

In graphics, datatypes can be very long. We can alias the original datatype using the `auto` keyword

```
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
  - A `const` variable means we cannot update it
  - A `const` function means we cannot modify any of the properties of that class

- Helps us by:
  - Telling the compiler that we don't need write access to the variable for faster optimization
  - Creating safer code by ensuring class variables won't be overwritten

- `degree()` will read the `_halfedge` of the `Face` class and iterates through each read-only halfedge to get the degree of the face

```
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 = 15362;
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 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`). These lead to different behaviors.

Const

- `HalfedgeCRef` is a `const_iterator`. That means that it is an iterator to a const value in the list
- We can change the address that `h` points to without accidentally changing the underlying data
  - Gives us a read-only copy

```
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);
```

- `static` defines a resource that is accessible in a larger scope than it exists
  - Inside of a class, we can define both functions and variables as `static`
- We can call them without creating an instance of the class by reference the `Mat4::` class-name

Static

- `static` variables are shared among all copies of object instances. All `Mat4` objects will always share the same `I` (identity) and `Zero` (zero matrix) values

```
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}};
```

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` encapsulate data and types into a group for easier referencing
- Helps us avoid conflicts if we have variables and functions of the same name in different places
- In Scotty3D we use `namespace` to group together different pieces of the graphics pipeline
  - In A3, we use `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;

    };

}
```

- The `Tri_Mesh` class is enclosed in the namespace `PT`
- To reference an instance, 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;

    };

}
```

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

```
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
- **Ex:** 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
```

- `a->undo()` calls the derived instance
- `a->redo()` calls the base instance

- `virtual` functions is good for runtime polymorphism
  - Common when you want to use a base class 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
- This example requires supporting `undo()` and `redo()` operations so that all operations have a valid way of being undone and redone

Datatypes

- The `vector` class is very common! Meant to handle random-access iterators well
- 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, 15362);               // create int vector of size 10 initialized as 15362
a.size();                               // returns 50
a.push_back(15362);                     // adds 15362 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)(*p);
 p++;}
```

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 via allocations and free
- **Ex:** 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

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 adding a child joint to our skeleton, we call the `Joint(unsigned int id, Joint* parent, Vec3 extent)` constructor using the `new` keyword to allocate an instance

```
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;
};
```

- 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. Any modifications to the variable are binded to the function's local-scope
- **Pass-by-reference** passes the address of the variable in memory. Any modifications are reflected 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`: any modifications will be local in scope to the function
  - `pt`: any modifications will be saved outside the function
- **Useful when we want to return multiple data:** we return a bool and the hit point `pt`

- ~~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 :)
  -- 15362 Staff