Game Architecture
Serialization

Today’s Agenda

- What is serialization and why?
- Methods and types of serialization.
- Reflection.

What is serialization?

- The deconstruction of an object or data structure into a byte stream.
- Reversible.

What is serialization?

- **The deconstruction of an object into a byte stream.**
- Reversible.

A byte stream could be anything. Commonly:

- A file on the disk.
- A socket.

What is serialization?

- The deconstruction of an object into a byte stream.
- **Reversible.**

Not a very useful operation unless we can reconstruct our original data.

Exception: Plot device on Star Trek

Why?

- Save Game

- Inventory
- Experience
- Progress

Why?

- Save Game
- Game Data

- Levels
- Entity Archetypes
- Configurations

Why?

- Save Game
- Game Data
- Network Play

- Position changed
- Fired weapon
- Health changed

Categories

- Text
- Binary

One size does not fit all.

Categories

- **Text**
  - Easy to debug
  - No special tools required for offline editing.
  - Portable. No need to worry about hardware details like word size and endianess.
- Binary
  - JSON
  - XML
  - CSV

Categories

- Text
- **Binary**
  - Fast
  - Compact

Text Serialization

```
struct save_game {
     int _level;
     float _health;
};

void serialize(std::ostream& os, const save_game& s)
{
     os << s._level << s._health;
}

void deserialize(std::istream& is, save_game& s)
{
     is >> s._level >> s._health;
}
```

Binary Serialization

```
struct save_game {
     int _level;
     float _health;
};

void serialize(std::ostream& os, const save_game& s)
{
     os.write(reinterpret_cast<char*>(&s._level), sizeof(int));
     os.write(reinterpret_cast<char*>(&s._health), sizeof(float));
}

void deserialize(std::istream& is, save_game& s)
{
     is.read(reinterpret_cast<char*>(&s._level), sizeof(int));
     is.read(reinterpret_cast<char*>(&s._health), sizeof(float));
}
```

Example Usage

```
{     // Write out save data
      save_game s{3, 1.0f};
      std::ofstream out(“save.dat”, std::ios::binary);

      serialize(out, s);
}

// save.dat
00000000000000000000000000000100 // _level
00111111100000000000000000000000 // _health

{     // Read it back in
      save_game s

      std::ifstream in(“save.dat”, std::ios::binary);
      deserialize(in, s);

      assert(s._level == 3);
      assert(s._health = 1.0f);
}
```

Small Problem!

We’re repeating ourselves a lot.

Insight!

- There’s two fundamental parts to the problem of serialization.
- We’ve been trying to solve both at once!

1. Figure out which data we want to serialize
    - _level
    - _health
2. Decide how it should be read or written.
    - Binary
    - Text

An improvement

```
// The “how” of serialization
struct binary_output_archive
{
     binary_output_archive(const char* file) :
          _os(file, std::ios::binary)
     {}

     template<typename T>
     void operator()(const T &t)
     {
          _os.write(
               reinterpret_cast<char*>(&t),
               sizeof(T));
     }

     std::ofstream _os;
};
```

An improvement

```
// The “how” of serialization
struct binary_output_archive
{
     binary_output_archive(const char* file) :
          _os(file, std::ios::binary)
     {}

     template<typename T>
     void operator()(const T &t)
     {
          _os.write(
               reinterpret_cast<char*>(&t),
               sizeof(T));
     }

     std::ofstream _os;
};
```

```
// Write out some data
{
     int x = 3;

     binary_output_archive out(“test.dat”);
     out(x);
}

// test.dat
00000000000000000000000000000011
```

An improvement

```
// The “how” of deserialization
struct binary_input_archive
{
     binary_input_archive(const char* file) :
          _is(file, std::ios::binary)
     {}

     template<typename T>
     void operator()(T& t)
     {
          _is.read(
               reinterpret_cast<char*>(&t),
               sizeof(T));
     }

     std::ifstream _is;
};
```

```
// Write out some data
{
     int x = 3;

     binary_output_archive out(“test.dat”);
     out(x);
}

// test.dat
00000000000000000000000000000011

// Read in some data
{
     int x;

     binary_input_archive in(“test.dat”);
     in(x);

     assert(x == 3);
}
```

An improvement

```
// The “what” of deserialization
template<typename Archive>
void serialize(Archive& ar, save_game& s)
{
     ar(s._level);
     ar(s._health);
}
```

An improvement

```
// The “what” of deserialization
template<typename Archive>
void serialize(Archive& ar, save_game& s)
{
     ar(s._level);
     ar(s._health);
}
```

```
{    // Write out save data
     save_game s{3, 1.0f};

     binary_output_archive out(“save.dat”);
     serialize(out, s);
}

// save.dat
00000000000000000000000000000100 // _level
00111111100000000000000000000000 // _health

{    // Read it back in
     save_game s;

     binary_input_archive in(“save.dat”);
     serialize(in, s);

     assert(s._level == 3);
     assert(s._health == 1.0f);
}
```

Problem?

This is *still* a bit tedious.

Solution

```
template <typename Archive, typename T>
void serialize(Archive& archive, const T& data)
{
     if (std::is_class<T>::value)
     {
          for (magic::field f : magic::get_fields(data))
          {
               serialize(archive, f);
          }
     }
     else
     {
          archive(data);
     }
}
```

Reflection

The ability of a computer program to examine, introspect, and modify its own structure and behavior

We’re mostly interested in the introspection part.

Reflection is not just for serialization!

- Logging
- Tools
- Duck Typing
- You’ll find other uses

Reflection in C++

- Ad-hoc
- Manual
- Attribute Parsing
- Type Definition Language

Reflection in C++

- **Ad-hoc**
- Manual
- Attribute Parsing
- Type Definition Language

This is essentially what we've been doing. We specify the data we're interested in, in our serialize methods.

- Pros
  - Dead simple.
  - Might be all you need for small games.
- Cons
  - Very brittle
  - Need to keep serialize method in sync with data.
  - Only useful for serialization.

Reflection in C++

- Ad-hoc
- **Manual**

```
class MyType {
	Vector2 _pos;
	std::string _name;
	void update();
};
reflect::type<MyType>("MyType").
	field("pos, &MyType::_pos).
	field("name", &MyType::_name).
	method("update", &MyType::update);
```

- Attribute Parsing
- Type Definition Language

Write it by hand.

- Pros
  - Easy to implement.
  - The type information is separate from the serialization code and can be used anywhere.
- Cons
  - The reflection data still needs to be updated as the type changes.

Reflection in C++

- Ad-hoc
- Manual
- **Attribute Parsing**

```
/// [[reflect]]
class MyType {
	/// [[field]]
	Vector2 _pos;
	/// [[field]]
	std::string _name;
	/// [[method]]
	void update();
};
```

- Type Definition Language

Add a pre-build step to parse the attributes.

- Pros
  - Unobtrusive
  - Reflection data available outside of the game (i.e. can be used by tools).
- Cons
  - Longer build times

Reflection in C++

- Ad-hoc
- Manual
- Attribute Parsing
- **Type Definition Language**

```
<type name=”MyType”>
  <field type=”Vector2” name=”pos”>
  <field type=”string” name=”name”>
  <method return_type=”void” name=”update”>
</type>
```

Write a tool that generates the C++ code and the reflection data from a description.

- Pros
  - No build time overhead.
    - Tool only needs to run when the type definitions change.
  - Reflection data available anywhere.
- Cons
  - Definitions and game out of sync.
  - Generated code will confuse all of your development tools.
    - Debugging/Breakpoints
    - IntelliSense/Autocomplete

Custom Type Definition Language

```
[Serialized]
class ga_foo
{
     uint32_t _bitfield;

     [Category("Hidden"), AutoGet, AutoSet(Private), BitField("_bitfield")]
     [InvarianceFunction("is_enabled_changed")]
     bool _is_enabled : 1 = true;

     [AutoGet, AutoSet(Private), BitField("_bitfield")]
     int _enable_count : 4 = 0;

public:
     virtual void is_enabled_changed() {
          set_enable_count(get_enable_count() + (get_is_enabled() ? 1 : -1);
     }
};
```

Custom Type Definition Language

```
class ga_foo
{
     uint32_t _bitfield;
public:
     inline void set_is_enabled(bool enabled) {
          BITFIELD_SET(
               _bitfield, enabled, 0, 1, bool);
          is_enabled_changed(this);
     }
     inline int get_enable_count() {
          return BITFIELD_GET(
               _bitfield, 1, 5, int);
     }
     virtual void is_enabled_changed() {
          set_enable_count(get_enable_count() + (get_is_enabled() ? 1 : -1);
     }
};
```

Versioning

Versioning

- Tolerance of unexpected data.
- Tolerance of missing data.

Versioning

- **Tolerance of unexpected data.**

```
// Runtime 1.0
struct save_game {
     int _level;
     int _difficulty;
     float _health;
};

// Version 2.0
struct save_game {
     int _level;
     float _health;
};
```

- Tolerance of missing data.

Versioning

- **Tolerance of unexpected data.**

```
// Runtime 1.0
struct save_game {
     int _level;
     int _difficulty;
     float _health;
};

// Version 2.0
struct save_game {
     int _level;
     float _health;
};
```

- Tolerance of missing data.

```
struct binary_output_archive {
     binary_output_archive(
          const char* file) :
               _os(file, std::ios::binary)
     {
     }

     template<typename T>
     void operator()(const T &t){
          _os.write(
               reinterpret_cast<char*>(&t),
               sizeof(T));
     }

     std::ofstream _os;
};
```

Versioning

- **Tolerance of unexpected data.**

```
// Runtime 1.0
struct save_game {
     int _level;
     int _difficulty;
     float _health;
};

// Version 2.0
struct save_game {
     int _level;
     float _health;
};
```

- Tolerance of missing data.

```
struct binary_output_archive {
     binary_output_archive(
          const char* file,
          int version = 0) :
               _os(file, std::ios::binary),
               _version(version)
     {
          // Write the version header
          (*this)(_version);
     }

     template<typename T>
     void operator()(const T &t){
          _os.write(
               reinterpret_cast<char*>(&t),
               sizeof(T));
     }

     std::ofstream _os;
     int _version;
};
```

Versioning

- **Tolerance of unexpected data.**

```
// Runtime 1.0
struct save_game {
     int _level;
     int _difficulty;
     float _health;
};

// Version 2.0
struct save_game {
     int _level;
     float _health;
};
```

- Tolerance of missing data.

```
struct binary_input_archive {
     binary_input_archive(const char* file) :
          _is(file, std::ios::binary)
     {
     }

     template<typename T>
     void operator()(T& t){
          _is.read(
               reinterpret_cast<char*>(&t),
               sizeof(T));
     }

     std::ifstream _is;
};
```

Versioning

- **Tolerance of unexpected data.**

```
// Runtime 1.0
struct save_game {
     int _level;
     int _difficulty;
     float _health;
};

// Version 2.0
struct save_game {
     int _level;
     float _health;
};
```

- Tolerance of missing data.

```
struct binary_input_archive {
     binary_input_archive(const char* file) :
          _is(file, std::ios::binary)
     {
          // Read the version header
          (*this)(_version);
     }

     template<typename T>
     void operator()(T& t){
          _is.read(
               reinterpret_cast<char*>(&t),
               sizeof(T));
     }

     std::ifstream _is;
     int _version;
};
```

Versioning

- **Tolerance of unexpected data.**

```
// Version 1.0
struct save_game {
     int _level;
     int _difficulty;
     float _health;
};

// Version 2.0
struct save_game {
     int _level;
     float _health;
};
```

- Tolerance of missing data.

```
// Write out Version 1.0 save data
{
     save_game s{3, 4, 1.0};

     binary_output_archive out("save.dat", 1);
     serialize(out, s);
}

// save.dat
00000000000000000000000000000001 // Version
00000000000000000000000000000100 // _level
00000000000000000000000000000011 // _difficulty
00111111100000000000000000000000 // _health

// Read it back in
{
     save_game s;

     binary_input_archive in("save.dat");
     serialize(in, s);

     assert(s._level == 3);
     assert(s._difficulty == 4);
     assert(s._health == 1.0f);
}
```

Versioning

- **Tolerance of unexpected data.**

```
// Version 1.0
struct save_game {
     int _level;
     int _difficulty;
     float _health;
};

// Version 2.0
struct save_game {
     int _level;
     float _health;
};
```

- Tolerance of missing data.

```
// Write out Version 2.0 save data
{
     save_game s{3, 1.0f};

     binary_output_archive out("save.dat", 2);
     serialize(out, s);
}

// save.dat
00000000000000000000000000000010 // Version
00000000000000000000000000000100 // _level
00111111100000000000000000000000 // _health

// Read it back in
{
     save_game s;

     binary_input_archive in("save.dat");
     serialize(in, s);

     assert(s._level == 3);
     assert(s._health == 1.0f);
}
```

Versioning

- **Tolerance of unexpected data.**

```
// Version 1.0
struct save_game {
     int _level;
     int _difficulty;
     float _health;
};

// Version 2.0
struct save_game {
     int _level;
     float _health;
};
```

- Tolerance of missing data.

```
// Version 1.0
template<typename Archive>
void serialize(Archive& ar, save_game& s)
{
     assert(ar._version == 1);

     archive(s._level);
     archive(s._difficulty);
     archive(s._health);
}

// Version 2.0
template<typename Archive>
void serialize(Archive& ar, save_game& s)
{
     archive(s._level);
     if (ar._version < 2) {
          int dummy_difficulty = 0;
          archive(dummy_difficulty);
     }
     archive(s._health);
}
```

Versioning

- Tolerance of unexpected data.
- **Tolerance of missing data.**

```
// Version 2.0
struct save_game {
     int _level;
     float _health;
};

// Version 3.0
struct save_game {
     int _level;
     int _difficulty;
     float _health;
};
```

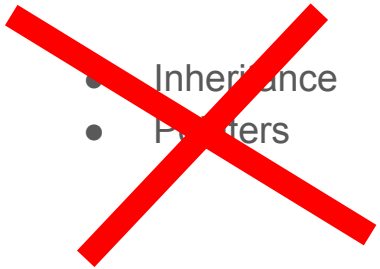
```
// Version 2.0
template<typename Archive>
void serialize(Archive& ar, save_game& s)
{
     archive(s._level);
     if (ar._version < 2) {
          int dummy_difficulty = 0;
          archive(dummy_difficulty);
     }
     archive(s._health);
}

// Version 3.0
template<typename Archive>
void serialize(Archive& ar, save_game& s)
{
     archive(s._level);
     if (ar._version == 2) {
          // No difficulty in v2. Default easy.
          s._difficulty = 0;
     }
     else {
          archive(s._difficulty);
     }
     archive(s._health);
}
```

Other Complications

- Inheritance
- Pointers

Other Complications

- Inheritance
- Pointers



Questions?