SERIALIZATION

FOR GAM 150 CLUB

Randy Gaul

Who am I?

- RTIS Sophomore – Randy Gaul
- C game as Freshman
- Tech director for Ancient Forest and Grumpy Monsters
- Made engine in C during summer before Sophomore year
  - AsciiEngine – Implemented some features in this slideshow
- Love architecture with clean and powerful APIs

Serialization - Overview

- The problem
- Primitives
- Routines
- Improvement
- Generic serialization
  - Crazy happens

The Problem

- So now we have game objects
- However, we need to:
  - Save our game state
  - Close application
  - Reload previous save

The solution

- Serialization
  - Translating data structures into a file format, to be later “resurrected” back into data structures.
- Useful for:
  - Saving and loading
  - Level files
  - Archetypes

Primitives

- Keep it simple
- Write serialization routines for primitives
- Serialize complex objects with primitive routines
- Handling serialization in two layers like this:
  - Breaks problem down into simpler steps
  - Results in more intuitive implementation

Primitives

- Serializing primitives:
  - SerializeFloat
  - SerializeInt
  - SerializeString

```
void SerializeFloat( float f, FILE *fp );
void SerializeInt( int i, FILE *fp );
void SerializeString( const char *s, FILE *fp );
```

Primitives - Implementation

```
void SerializeFloat( float f, FILE *fp )
{
  fprintf_s( fp, "%f\n", f );
}

void SerializeInt( int i, FILE *fp )
{
  fprintf_s( fp, "%d\n", i );
}

void SerializeString( const char *s, FILE *fp )
{
  fprintf_s( fp, "%s\n", s );
}
```

Primitives Deserialization

- Deserialization is reading data to make something

```
void DeserializeFloat( float *f, FILE *fp )
{
  fscanf_s( fp, "%f\n", f );
}

void DeserializeInt( int *i, FILE *fp )
{
  fscanf_s( fp, "%d\n", i );
}

void DeserializeString( const char *s, FILE *fp )
{
  fscanf_s( fp, "%s\n", s );
}
```

Routines by Primitives

- Combine primitives to make something more complex

```
typedef struct Object
{
  char *id;
  int x, y;
  float rotation;
} Object;
```

```
void SerializeObject( Object *o, FILE *fp )
{
  SerializeString( o->id );
  SerializeInt( &o->x ); SerializeInt( &o->y );
  SerializeFloat( &o->rotation );
}
```

Routines by Primitives

- Deserialization of Object

```
typedef struct Object
{
  char *id;
  int x, y;
  float rotation;
} Object;
```

```
void DeserializeObject( Object *o, FILE *fp )
{
  DesrializeString( o->id );
  DeserializeInt( &o->x ); DeserializeInt( &o->y );
  DeserializeFloat( &o->rotation );
}
```

Serialization

- Output file:

```
ObjectIdentifier
6
7
0.142
```

- Magic numbers
  - Lets clean this up
  - We want our file to be prettier
    - Easier to modify and read by hand

Questions?

Serialization - Prettify

- Slightly prettier output:

```
void SerializeObject( Object *o, FILE *fp )
{
  f_printf( fp, "{\n" );
  f_printf( fp, "  " ); SerializeString( o->id );
  f_printf( fp, "  " ); SerializeInt( &o->x );
  f_printf( fp, "  " ); SerializeInt( &o->y );
  f_printf( fp, "  " ); SerializeFloat( &o->rotation );
  f_printf( fp, "}\n" );
}
```

```
{
  ObjectIdentifier
  6
  7
  0.142
}
```

Serialization - Prettify

- A little better, but what about:

```
typedef struct BiggerObject
{
  Object obj;
  char *messsage;
} BiggerObject;
```

```
void SerializeBiggerObject( BiggerObject *obj, FILE *fp )
{
  f_printf( fp, "{\n" );
  f_printf( fp, "  " ); SerializeObject( (Object *)obj, fp );
  f_printf( fp, "  " ); SerializeString( obj->message, fp );
  f_printf( fp, "}\n" );
}
```

Serialization - Prettify

- We now have a problem with our “prettiness”

```
{
  {
  ObjectIdentifier
  6
  7
  0.142
}
  TheMessageString
}
```

- Our serialization is not modular

Serialization - Prettify

- A working “prettiness” method
  - Able to adjust padding height as we go along

```
void Padding( int increment, FILE *fp )
{
  static int pad_level = 0;
  unsigned i = 0;
  pad_level += increment;

  while(i < pad_level)
  {
    fprintf( "   " );
    ++i;
  }
}
```

Serialization - Prettify

- New routine for object:

```
void SerializeObject( Object *o, FILE *fp )
{
  Padding( 0, fp );
  fprint( fp, "{\n" );
  Padding( 1, fp ); SerializeString( o->id );
  Padding( 0, fp ); SerializeInt( &o->x );
  Padding( 0, fp ); SerializeInt( &o->y );
  Padding( 0, fp ); SerializeFloat( &o->rotation );
  Padding( -1, fp ); f_printf( fp, "}\n" );
}
```

- Control pad height during routine
  - Only care about how much we increment

Serialization - Prettify

- Fixed output for BiggerObject
  - Assuming we used padding in BiggerObject too

```
{
  {
    ObjectIdentifier
    6
    7
    0.142
  }
  TheMessageString
}
```

Serialization – Cleanup API

- Lets cleanup all these padding calls
- Rewrite our pad function

```
ChangePaddingLevel( int increment ) {
  Padding( increment, NULL );
}

void Padding( int increment, FILE *fp ) {
  static int pad_level = 0;
  unsigned i = 0;
  pad_level += increment;

  if(fp) PlacePads( pad_level, fp );
}

void PlacePads( int level, FILE *fp ) {
  unsigned i = 0;
  while(i < level) {
    fprintf( "  " );
    ++i;
  }
}
```

Serialization – Cleanup API

- Cleaner version of SerializeObject
  - Use new pad functions
  - Call Padding(0, fp) in primitives

```
void SerializeObject( Object *o, FILE *fp )
{
  SerializeString( "{", fp );
  ChangePaddingLevel( 1 );
  SerializeString( o->id );
  SerializeInt( &o->x );
  SerializeInt( &o->y );
  SerializeFloat( &o->rotation );
  ChangePaddingLevel( -1 );
  SerializeString( "}", fp );
}
```

Serialization – Cleanup API

- Cleaner version of serializing BiggerObject

```
void SerializeBiggerObject( BiggerObject *obj, FILE *fp )
{
  SerializeString( "{", fp );
  ChangePaddingLevel( 1 );
  SerializeObject( (Object *)obj, fp );
  SerializeString( obj->message, fp );
  ChangePaddingLevel( -1 );
  SerializeString( "}", fp );
}
```

- Redundancy:
  - We always call:
    - SerializeString(“{“) and increment pads by 1
    - Same with closing bracket and -1

Serialization – Cleanup API

- Even cleaner version:

```
void DeserializeBiggerObject( BiggerObject *obj, FILE *fp )
{
  OpenBracket( fp );
  SerializeObject( (Object *)obj, fp );
  SerializeString( obj->message, fp );
  CloseBracket( fp );
}
```

- Open bracket:
  - Increment pad level, place pads
- Close bracket:
  - Decrement pad level, place pads

Serialization – Cleanup API

- One problem:
  - Our prettification from left to right introduced brackets

```
ObjectIdentifier
6
7
0.142
```

```
{
  ObjectIdentifier
  6
  7
  0.142
}
```

- Deserialization does not expect brackets!

Serialization – Cleanup API

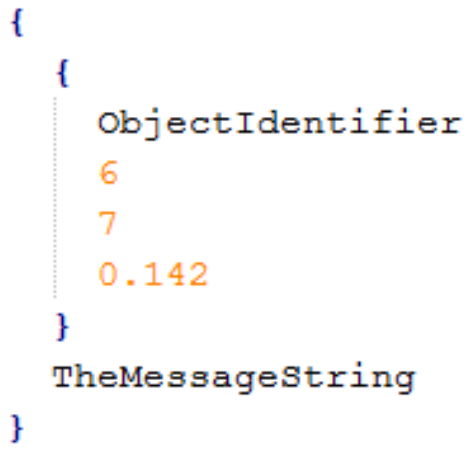
- Compensate with “eat bracket” function

```
void DeserializeObject( Object *o, FILE *fp )
{
  EatBrackets( fp );
  DeserializeString( o->id );
  DeserializeInt( &o->x );
  DeserializeInt( &o->y );
  DeserializeFloat( &o->rotation );
  EatBrackets( fp );
}
```

- EatBrackets:
  - Use fgetc until you find ‘{‘ or ‘}’, then eat newline
    - Or use scanf power – mentioned in later slide

Serialization

- Final “prettified output” of BiggerObject



- Quite nice! Still room to improve:
  - Name of object?
  - Name of each data member?

Generic Serialization

- Lets see if we can make this jump

```
{
  {
    ObjectIdentifier
    6
    7
    0.142
  }
  TheMessageString
}
```

```
BiggerObject
{
  Object
  {
    id = "ObjectIdentifier"
    x = 6
    y = 7
    rotation = 0.142
  }
  message = "TheMessageString"
}
```

- Things are going to get crazy.
  - Lets automate serialization so we write only a single routine for all structs!

Questions?

Generic Serialization

- scanf is mighty: link
- %*[flags] [width] [.precision] specifier
- []
  - Scanset; Reads chars as string until char is found that is not listed between []
- [^]
  - Negation scanset; same as above, but reads all except for chars between ^]
- *
  - Negation; read specifier and do not assign (skip text)

Generic Serialization

- Given a text file, read the float 5.50:

```
Text file
asdkfksdafjasd fasjk w i w
asdfja f-2 fn fdanf
13o31i f -a f = 5.50
```

- fscanf(fp, “%*[^=]= %f”, &var);
  - Read in a string until = if found, do not assign this string
  - Read in equal sign
  - Read in a float and assign to &var

Generic Serialization

- Some new primitives
  - Skip over text until = sign found, read in data

```
void Deserializeint( int *a, FILE *fp )
{
  fscanf( fp, "%*[^=]= %d", a );
}

void Deserializefloat( float *a, FILE *fp )
{
  fscanf( fp, "%*[^=]= %f", a );
}

void Deserializestring( const char *a, FILE *fp )
{
  fscanf( fp, "%*[^=]= %s", a ); // :)
}
```

Generic Serialization

- Okay prettification done
- Lets automate our serialization routine
- Write single routine for all structs
- But how?
  - Macros

A New Way of Thinking

- Tools at our disposal:
  - Parameterized macro
  - Stringize
  - Token pasting
  - Undef
  - Include protection
    - #pragma once

A New Way of Thinking

- Overall idea
  - Create data file that fills out some macros
  - Define those macros
  - Include data file
    - Macros expand
  - Undef macros
  - Define new macros
  - Inlcude data file
    - Macros expand to something else

A New Way of Thinking

- Data file

```
_NAME( MyObject )
_MEMBER( string, id )
_MEMBER( int, x )
_MEMBER( int, y )
_MEMBER( float, rotation )
_END( MyObject );
```

- MyObjectData.h
  - Holds inputs to yet to be defined macros

A New Way of Thinking

- Defines

```
#undef _NAME
#undef _MEMBER
#undef _END

#define _NAME( NAME ) \
  void Serialize##NAME( NAME *obj, FILE *fp ) \
  { \
    SerializeType( #NAME, fp ); \
    OpenBracket( fp );

#define _MEMBER( TYPE, MEMBER ) \
    SerializeMember( #MEMBER, fp ); \
    Serialize##TYPE( obj->MEMBER, fp );

#define _END( NAME ) \
    CloseBracket( fp ); \
  }
```

A New Way of Thinking

- The new MyObject.h

```
#pragma once

#include "DeclareStruct.h"
#include "MyObjectData.h"
#include "SerializationDeclare.h"
#include "MyObjectData.h"
```

- MyObject.c

```
#include "Precompiled.h"

#include "MyObject.h"
#include "SerializeDefine.h"
#include "MyObjectData.h"
#include "DeserializeDefine.h"
#include "MyObjectData.h"
```

A New Way of Thinking

- We're importing data from MyObjectData
- Macros expand to interpret the data:
  - Header file for MyObject
  - C file for MyObject
  - Serialization/Deserialization routine for MyObject
- We only wrote one data file
  - We generated 3 files from 1
  - Updates to data file automatically propagate to all macro expansions

A New Way of Thinking

- Lest step through it
- Data file ----------→
- Macros:

```
_NAME( MyObject )
_MEMBER( string, id )
_MEMBER( int, x )
_MEMBER( int, y )
_MEMBER( float, rotation )
_END( MyObject );
```

```
#undef _NAME
#undef _MEMBER
#undef _END

#define _NAME( NAME ) \
  typedef struct NAME{

#define _MEMBER( TYPE, MEMBER ) \
  TYPE MEMBER;

#define _END( NAME ) \
  } NAME;
```

A New Way of Thinking

```
_NAME( MyObject )
_MEMBER( string, id )
_MEMBER( int, x )
_MEMBER( int, y )
_MEMBER( float, rotation )
_END( MyObject );
```

```
typedef struct MyObject{
string id;
int x;
int y;
float rotation;
_END( MyObject );
```

```
typedef struct MyObject{
_MEMBER( string, id )
_MEMBER( int, x )
_MEMBER( int, y )
_MEMBER( float, rotation )
_END( MyObject );
```

```
typedef struct MyObject{
string id;
int x;
int y;
float rotation;
} MyObject;
```

```
typedef struct MyObject{
string id;
_MEMBER( int, x )
_MEMBER( int, y )
_MEMBER( float, rotation )
_END( MyObject );
```

Generic Serialization

- Summary:
- Three files
  - Data
  - Header
  - C
- We get:
  - Free serialization routines
  - Automatically updated when data file changes
- Sample code for proof of concept: link

Generic… Other things?

- Other things can be generalized:
  - Scripting language integration
    - Best thing ever
  - Factories
  - Generic “Variable” type
  - Property grid generation
  - More?
- Serialization is enough for now

Final Tips

- Cannot pass pointer type to macro
  - typedef
  - typedef char * string
    - Pass string as type to macro
- Ask Doug Schilling for advice! He's awesome
- Keep things as simple as you can
- Ask upper classmen questions
  - Email me: r.gaul@digipen.edu

Resources:

- https://github.com/RandyGaul/Serialization_C
- http://www.randygaul.net/2013/02/07/fscanf-power/
- http://www.randygaul.net/2012/08/10/generic-programming-in-c/

Questions

- Anybody have ‘em?