UD1434 - Litet Spelprojekt Exporter/Importer

David Lyhed Danielsson

dld@bth.se

Today

- Project introduction
- Memory in C++
- Binary files
  - Reading
  - Writing

Course plan

Kod	Benämning	Omfattning	Betyg
1505	Projektredovisning	1.5 hp	G-U
1515	Projekt	9 hp	G-U
1525	Exporter- importer programvara	4 hp	G-U
1535	Redovisning exporter- importer	0.5 hp	G-U

Goals for this part of the course

- Understand game assets management
  - How is data transformed and represented
  - How do requirements affect the pipeline
  - How to compare different alternatives
    - Performance vs size
    - Size vs precision
  - How do different compression approaches affect the transformation pipeline
    - Lossless vs lossy
    - HW support for compressed textures

Goals for this part of the course (2)

- Division of work from a SW perspective
  - Define early requirements and specs
    - Argue for them!
  - Implement different parts in parallel
    - Clear interfaces (headers)
    - Library stubs for quick division of work

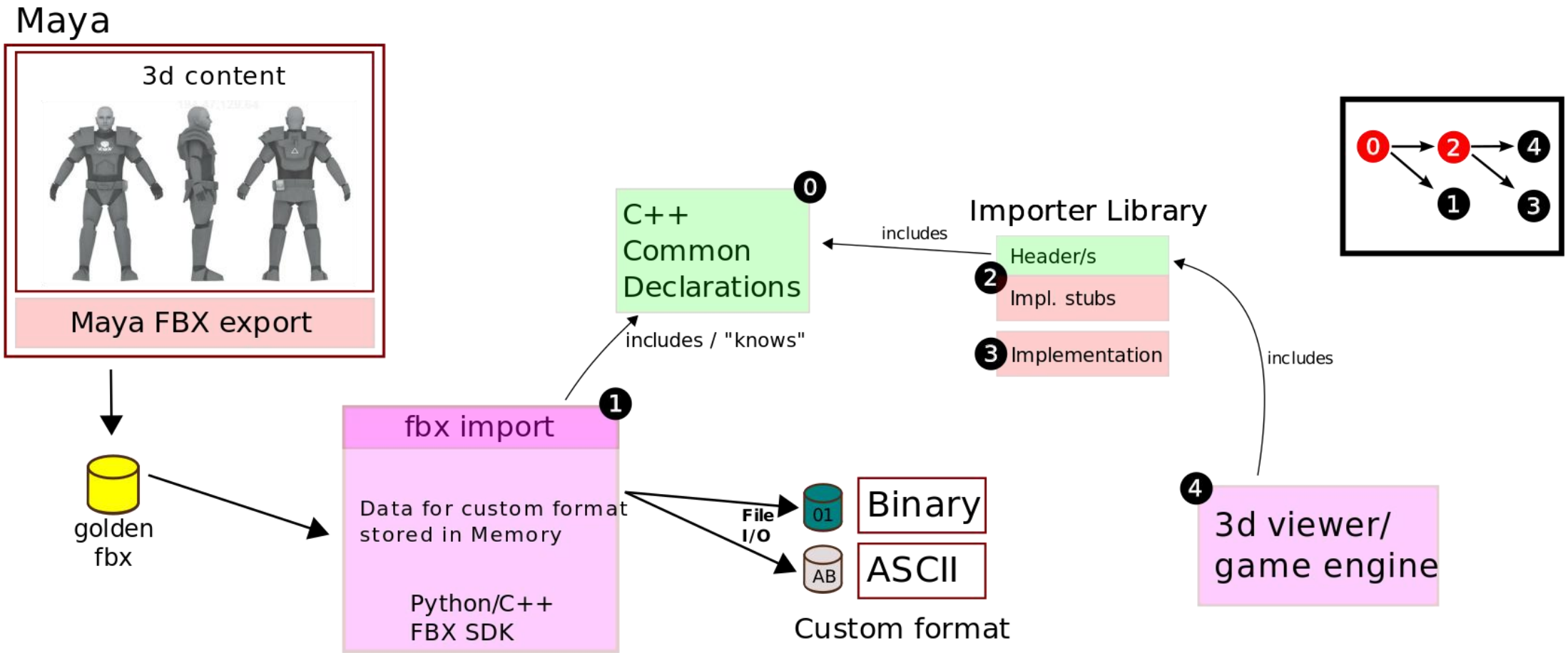
Goals for this part of the course (3)

- Technical aspects
  - Binary data layout and representation
  - Binary and text representation of data in file
  - C++ or Python FBX SDK API
  - Libraries (static)
  - Edit/compile/link cycle

Assignment - 4 hp + 0.5 hp

- Write an importer/exporter for the FBX format using the FBX SDK
- Write a small library to import a custom format into the game project
- Support binary and text custom format

Assignment - Big picture



Assignment workload

- **Tier 1 (1 student per group)**
  - FBX conversion to custom binary, in C++ or Python FBX API
  - Custom format importer in C++ (library - .cpp and .h files)
  - Vertex information
    - Position
    - Texture coordinates
    - Normals, tangents, bi-tangents
  - Materials
    - Textures: embedded or copied to the same directory as the custom format
- **Tier 2 (2 to 3 students per group) - All tier 1 features, plus:**
  - All scene camera
  - Morph-animation (all key frames on the vertex information)
  - All light sources
- **Tier 3 (4 students per group) - All tier 2 features, plus:**
  - Skeleton animations (all key frames)
  - Vertex weights for skinning (maximum 4, normalize if needed)
- **Tier 4 (5+ students per group) - All tier 3 features, plus**
  - Groups (empty transformations in a scene graph)
  - Custom attributes from Maya

If you do not want to support a feature in the game project, your custom format still has to support the features you are meant to implement!

Questions?

Binary files

- Raw data (memory) is written instead of text
- Raw data does not make sense in and of itself
- Data has to be interpreted in a known (well-defined) way
- To understand binary files, we must understand how computer memory works

Radix (base) recap

Binary

- Expressed in base 2
- Often prefixed 0b
- Counting to 5: 0, 1, 10, 11, 100, 101
- Maximum value = $2^{\text{digits}} - 1$
- 3 digits => $2^3 - 1 = 7$
- 111 = 7

Hexadecimal

- Expressed in base 16
- Often prefixed 0x
- Counting to 5: 0, 1, 2, 3, 4, 5
- Counting from 8 to 17: 8, 9, A, B, C, D, E, F, 10, 11
- Usually used to represent binary data

Computer memory in C++

- Binary
- 1 bit = a single 0 or 1 (digit)
  - 0b1 = 1 bit
  - 0b1011 = 4 bits
- 1 byte = 8 bits (max value of 2^8 - 1 = 255)
  - 0b00101101 = 8 bits (1 byte)
- Can be seen as a large consecutive array of bytes (1GiB = 1073741824 bytes)

Computer memory in C++ (2)

- Each primitive data type has an (almost) set type
- Size is always measured in bytes
- int is usually 32 bits (4 bytes) in x86 and 64 bits (8 bytes) in x64
- When working with binary files, using (u)intX_t types is the safest bet (defined in cstdint) for integer types
- https://msdn.microsoft.com/en-us/library/s3f49ktz.aspx
  - __intX windows specific, intX_t cross-platform

Type	Size	Value Range
bool	1	true/false
char	1	-128 to 127
short	2	-32768 to 32767
int	4	-2'147'483'648 to 2'147'483'647
int8_t	1	-128 to 127
uint8_t	1	0 to 255
int16_t	2	-32'768 to 32'767
uint16_t	2	0 to 65'535
int32_t	4	-2'147'483'648 to 2'147'483'647
uint32_t	4	0 to 4'294'967'295
int64_t	8	-9'223'372'036'854'775'808 to 9'223'372'036'854'775'807
uint64_t	8	0 to 18'446'744'073'709'551'615
float	4	$3.4*10^{\pm 38}$
double	8	$1.7*10^{\pm 308}$

Computer memory in C++ (3)

- `sizeof(int8_t) == 1` (8 bits)
- `sizeof(int16_t) == 2` (16 bits)
- 12345 = 0b0011000000111001
- Memory is an array of bytes, but some (most) types are larger than a byte

Computer memory in C++ - fitting a short into a byte array

- 12345 = 0b0011000000111001
- Split binary representation into byte-sized chunks
- Endianness matters - though probably not in this assignment

Binary files - Interpretation

- Data layout is defined by the programmers (you)
- Layout is tailor-made for the application; there is no one-size-fits-all solution
- Data only makes sense if you know the layout
- Remember: size of long, int, short, can vary between architectures!

Binary files - Example

Data only makes sense if you know the layout!

vertices.hexdump

```
1    Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
2   00000000: 00 00 80 BF 00 00 80 3F 00 00 00 00 00 00 00 00    ...?...?........
3   00000010: 00 00 00 00 00 00 80 3F 00 00 00 00 00 00 00 00    .......?........
4   00000020: 00 00 80 3F 00 00 80 3F 00 00 00 00 00 00 00 00    ...?...?........
5   00000030: 00 00 00 00 00 00 80 3F 00 00 80 3F 00 00 00 00    .......?...?....
6   00000040: 00 00 80 BF 00 00 80 BF 00 00 00 00 00 00 00 00    ...?...?........
7   00000050: 00 00 00 00 00 00 80 3F 00 00 00 00 00 00 80 3F    .......?.......?
8   00000060: 00 00 80 3F 00 00 80 BF 00 00 00 00 00 00 00 00    ...?...?........
9   00000070: 00 00 00 00 00 00 80 3F 00 00 80 3F 00 00 80 3F    .......?...?...?
```

Binary files - Example

Data only makes sense if you know the layout!

vertices.hexdump

```
1    Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
2  00000000: 00 00 80 BF 00 00 80 3F 00 00 00 00 00 00 00 00  ...?...?........
3  00000010: 00 00 00 00 00 00 80 3F 00 00 00 00 00 00 00 00  .......?........
4  00000020: 00 00 80 3F 00 00 80 3F 00 00 00 00 00 00 00 00  ...?...?........
5  00000030: 00 00 00 00 00 00 80 3F 00 00 80 3F 00 00 00 00  .......?...?....
6  00000040: 00 00 80 BF 00 00 80 BF 00 00 00 00 00 00 00 00  ...?...?........
7  00000050: 00 00 00 00 00 00 80 3F 00 00 00 00 00 00 80 3F  .......?.......?
8  00000060: 00 00 80 3F 00 00 80 BF 00 00 00 00 00 00 00 00  ...?...?........
9  00000070: 00 00 00 00 00 00 80 3F 00 00 80 3F 00 00 80 3F  .......?...?...?
```

- Position (3 floats)

Binary files - Example

Data only makes sense if you know the layout!

vertices.hexdump

```
1    Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
2  00000000: 00 00 80 BF 00 00 80 3F 00 00 00 00 00 00 00 00  ...?...?........
3  00000010: 00 00 00 00 00 00 80 3F 00 00 00 00 00 00 00 00  .......?........
4  00000020: 00 00 80 3F 00 00 80 3F 00 00 00 00 00 00 00 00  ...?...?........
5  00000030: 00 00 00 00 00 00 80 3F 00 00 80 3F 00 00 00 00  .......?...?....
6  00000040: 00 00 80 BF 00 00 80 BF 00 00 00 00 00 00 00 00  ...?...?........
7  00000050: 00 00 00 00 00 00 80 3F 00 00 00 00 00 00 80 3F  .......?.......?
8  00000060: 00 00 80 3F 00 00 80 BF 00 00 00 00 00 00 00 00  ...?...?........
9  00000070: 00 00 00 00 00 00 80 3F 00 00 80 3F 00 00 80 3F  .......?...?...?
```

- Position (3 floats)
- Normal (3 floats)

Binary files - Example

Data only makes sense if you know the layout!

vertices.hexdump

```
1    Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
2  00000000: 00 00 80 BF 00 00 80 3F 00 00 00 00 00 00 00 00  ...?...?........
3  00000010: 00 00 00 00 00 00 80 3F 00 00 00 00 00 00 00 00  .......?........
4  00000020: 00 00 80 3F 00 00 80 3F 00 00 00 00 00 00 00 00  ...?...?........
5  00000030: 00 00 00 00 00 00 80 3F 00 00 80 3F 00 00 00 00  .......?...?....
6  00000040: 00 00 80 BF 00 00 80 BF 00 00 00 00 00 00 00 00  ...?...?........
7  00000050: 00 00 00 00 00 00 80 3F 00 00 00 00 00 00 80 3F  .......?.......?
8  00000060: 00 00 80 3F 00 00 80 BF 00 00 00 00 00 00 00 00  ...?...?........
9  00000070: 00 00 00 00 00 00 80 3F 00 00 80 3F 00 00 80 3F  .......?...?...?
```

- Position (3 floats)
- Normal (3 floats)
- Texture coordinates (2 floats)

Binary files - Example

Data only makes sense if you know the layout!

vertices.hexdump

```
1    Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
2  00000000: 00 00 80 BF 00 00 80 3F 00 00 00 00 00 00 00 00  ...?...?........
3  00000010: 00 00 00 00 00 00 80 3F 00 00 00 00 00 00 00 00  .......?........
4  00000020: 00 00 80 3F 00 00 80 3F 00 00 00 00 00 00 00 00  ...?...?........
5  00000030: 00 00 00 00 00 00 80 3F 00 00 80 3F 00 00 00 00  .......?...?....
6  00000040: 00 00 80 BF 00 00 80 BF 00 00 00 00 00 00 00 00  ...?...?........
7  00000050: 00 00 00 00 00 00 80 3F 00 00 00 00 00 00 80 3F  .......?.......?
8  00000060: 00 00 80 3F 00 00 80 BF 00 00 00 00 00 00 00 00  ...?...?........
9  00000070: 00 00 00 00 00 00 80 3F 00 00 80 3F 00 00 80 3F  .......?...?...?
```

- Position (3 floats)
- Normal (3 floats)
- Texture coordinates (2 floats)

Binary files in C++

- Writing to a binary file is easy!
- Data is written as a series of bytes (char)
- Guaranteed to for any type that is trivially copyable
- Works as long as the class is continuous in memory
- Single function to write entire array - very fast!

vertices.hexdump

```
   Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 00 00 80 BF 00 00 80 3F 00 00 00 00 00 00 00 00   ...?...?........
00000010: 00 00 00 00 00 00 80 3F 00 00 00 00 00 00 00 00   .......?........
00000020: 00 00 80 3F 00 00 80 3F 00 00 00 00 00 00 00 00   ...?...?........
00000030: 00 00 00 00 00 00 80 3F 00 00 80 3F 00 00 00 00   .......?...?....
00000040: 00 00 80 BF 00 00 80 BF 00 00 00 00 00 00 00 00   ...?...?........
00000050: 00 00 00 00 00 00 80 3F 00 00 00 00 00 00 80 3F   .......?.......?
00000060: 00 00 80 3F 00 00 80 BF 00 00 00 00 00 00 00 00   ...?...?........
00000070: 00 00 00 00 00 00 80 3F 00 00 80 3F 00 00 80 3F   .......?...?...?
```

```cpp
#include <fstream>

int main()
{
    struct Vertex
    {
        float position[3];
        float normal[3];
        float uv[2];
    };

    const static int VERTEX_COUNT = 4;

    Vertex vertices[VERTEX_COUNT]
    {
        // posX  posY  posZ  norX  norY  norZ   U     V
        { -1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f }
        , { 1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 0.0f }
        , { -1.0f, -1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f }
        , { 1.0f, -1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f }
    };

    // Create binary file
    std::ofstream out("vertices", std::ios::binary);
    // Write entire vertices array to file
    // "vertices" is of type "Vertex", but can be represented as a char (byte) array
    // by breaking it into char sized chunks, and then be written to disk
    out.write(reinterpret_cast<char*>(vertices), sizeof(Vertex) * VERTEX_COUNT);
}
```

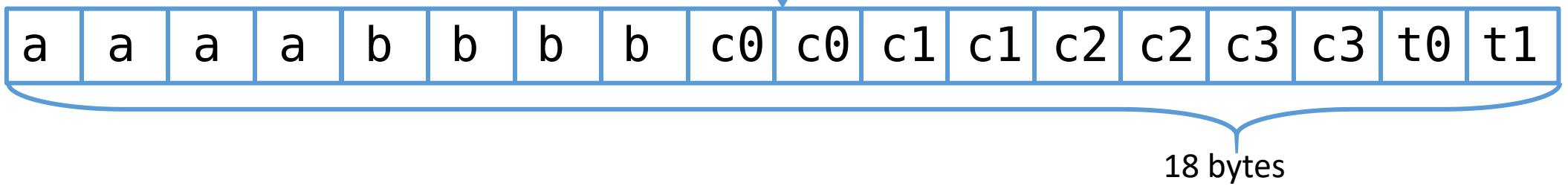
Binary files in C++ - Writing structs and classes

"Works as long as the class is continuous in memory"

- Different compilers produce different memory layouts
- This means no inheritance, no pointers, and no references
- It is safest to use structs/classes with only plain data

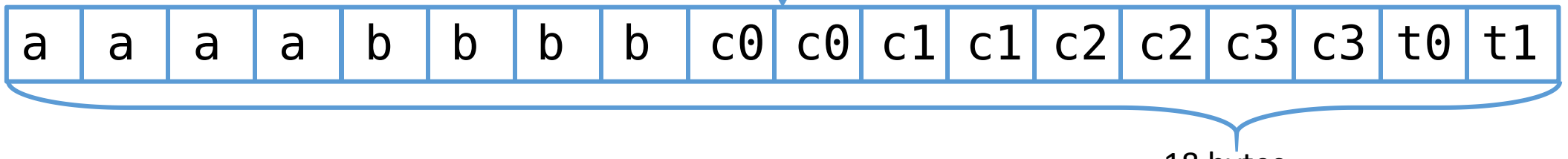
Continuous vs discontinuous memory

```
struct Continuous
{
    int32_t a;    // 4 bytes
    float b;      // 4 bytes
    int16_t c[4]  // 2 bytes * 4
    char t[2];    // 2 bytes
} // Total of 18 bytes
```



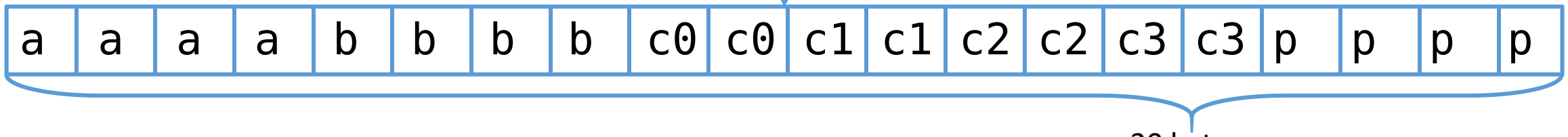
Continuous vs discontinuous memory

```
struct Continuous
{
    int32_t a;    // 4 bytes
    float b;      // 4 bytes
    int16_t c[4]  // 2 bytes * 4
    char t[2];    // 2 bytes
} // Total of 18 bytes
```



18 bytes

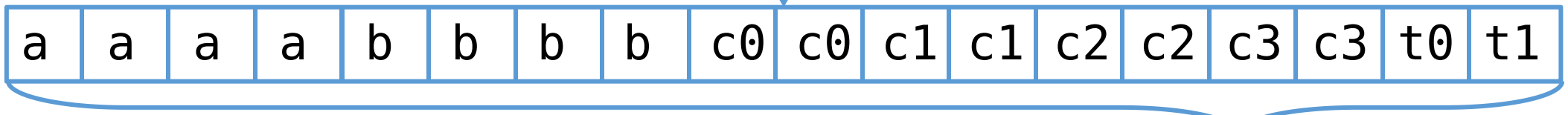
```
struct Discontinuous
{
    int32_t a;     // 4 bytes
    float b;       // 4 bytes
    int16_t c[4]   // 2 bytes * 4
    Continuous* p; // 4 bytes
} // Total of 20 bytes
```



20 bytes

Continuous vs discontinuous memory

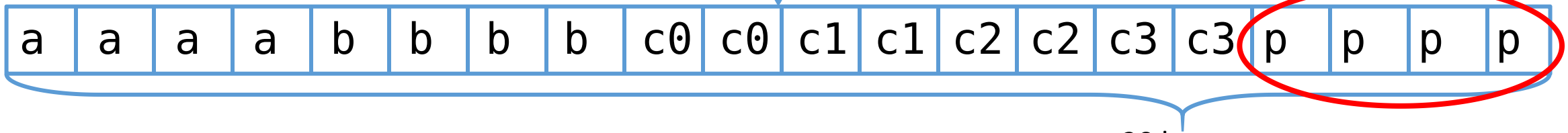
```
struct Continuous
{
    int32_t a;    // 4 bytes
    float b;      // 4 bytes
    int16_t c[4]  // 2 bytes * 4
    char t[2];    // 2 bytes
} // Total of 18 bytes
```



18 bytes

```
struct Discontinuous
{
    int32_t a;     // 4 bytes
    float b;       // 4 bytes
    int16_t c[4]   // 2 bytes * 4
    Continuous* p; // 4 bytes
} // Total of 20 bytes
```

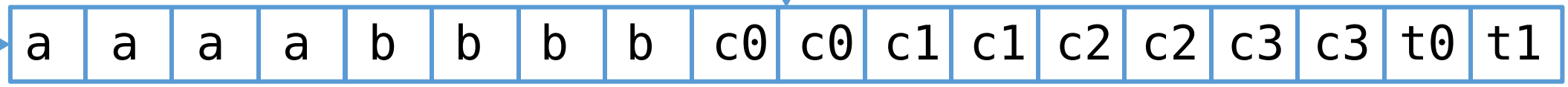
Pointer!



20 bytes

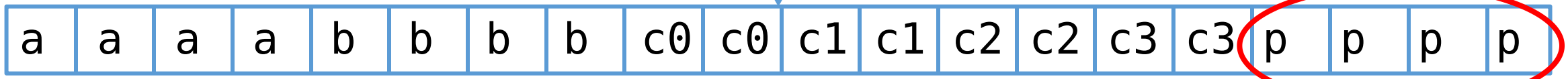
Continuous vs discontinuous memory

```
struct Continuous
{
    int32_t a;     // 4 bytes
    float b;       // 4 bytes
    int16_t c[4]   // 2 bytes * 4
    char t[2];     // 2 bytes
} // Total of 18 bytes
```




```
struct Discontinuous
{
    int32_t a;      // 4 bytes
    float b;        // 4 bytes
    int16_t c[4]    // 2 bytes * 4
    Continuous* p; // 4 bytes
} // Total of 20 bytes
```

Pointer!



`sizeof(Discontinuous) == 20`, but p points to Continuous, `sizeof(Continuous) == 18`

Continuous vs discontinuous memory - strings and other containers

- How big is an std::string?

Continuous vs discontinuous memory - strings and other containers

- How big is an std::string?

```
4   int main()
5   {
6       std::string emptyString;
7       std::string shortString = "Short";
8       std::string longString = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur accumsan ul
9
10      std::cout << "sizeof(emptyString) == " << sizeof(emptyString) << std::endl;
11      std::cout << "sizeof(shortString) == " << sizeof(shortString) << std::endl;
12      std::cout << "sizeof(longString) == " << sizeof(longString) << std::endl;
13  }
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                    1: /bin/bash

david@davidxubuntu:~/temp$ ./stringSize
sizeof(emptyString) == 32
sizeof(shortString) == 32
sizeof(longString) == 32
david@davidxubuntu:~/temp$
```

Continuous vs discontinuous memory - strings and other containers

- How big is an std::string?

```
4   int main()
5   {
6       std::string emptyString;
7       std::string shortString = "Short";
8       std::string longString = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur accumsan ul
9
10      std::cout << "sizeof(emptyString) == " << sizeof(emptyString) << std::endl;
11      std::cout << "sizeof(shortString) == " << sizeof(shortString) << std::endl;
12      std::cout << "sizeof(longString) == " << sizeof(longString) << std::endl;
13  }
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                    1: /bin/bash

david@davidxubuntu:~/temp$ ./stringSize
sizeof(emptyString) == 32
sizeof(shortString) == 32
sizeof(longString) == 32
david@davidxubuntu:~/temp$
```

- How do we write an std::string to a binary file if the size is always the same?
- Hint: discontinuous memory

Continuous vs discontinuous memory - strings and other containers (2)

- Each element in an std::string is the same size
- The string's text is guaranteed to be continuous in memory

1. How long is an std::string?
2. How big is each element in an std::string?
3. Where is the string's text data located?

Continuous vs discontinuous memory - strings and other containers (2)

- Each element in an std::string is the same size
- The string's text is guaranteed to be continuous in memory

1. How long is an std::string? `std::string::size()`
2. How big is each element in an std::string? `sizeof(char)`
3. Where is the string's text data located? `std::string::data()`

Continuous vs discontinuous memory - strings and other containers (2)

- Each element in an std::string is the same size
- The string's text is guaranteed to be continuous in memory

1. How long is an std::string? `std::string::size()`
2. How big is each element in an std::string? `sizeof(char)`
3. Where is the string's text data located? `std::string::data()`

- Writing the string's text is easy:

```
out.write(string.data(), string.size());
```

Continuous vs discontinuous memory - strings and other containers (2)

- Each element in an std::string is the same size
- The string's text is guaranteed to be continuous in memory

1. How long is an std::string? `std::string::size()`
2. How big is each element in an std::string? `sizeof(char)`
3. Where is the string's text data located? `std::string::data()`

- Writing the string's text is easy:

```
out.write(string.data(), string.size());
```

- std::vectors can also be written in this way:

```
out.write((char*)vector.data(), sizeof(vector[0]) * vector.size());
```

Binary files in C++ - Reading structs

- Not much different than writing
- Single function call to fill entire array with data - very fast!

```cpp
int main()
{
    const static int VERTEX_COUNT = 4;

    Vertex vertices[VERTEX_COUNT];

    // Open binary file
    std::ifstream in("vertices", std::ios::binary);
    // Read entire vertices array from file
    in.read(reinterpret_cast<char*>(&vertices), sizeof(Vertex) * VERTEX_COUNT);

    for(int i = 0; i < VERTEX_COUNT; ++i)
    {
        std::cout << "Vertex " << i + 1 << std::endl;
        PrintVertex(vertices[i]);
    }

    std::cin.get();
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: ./b

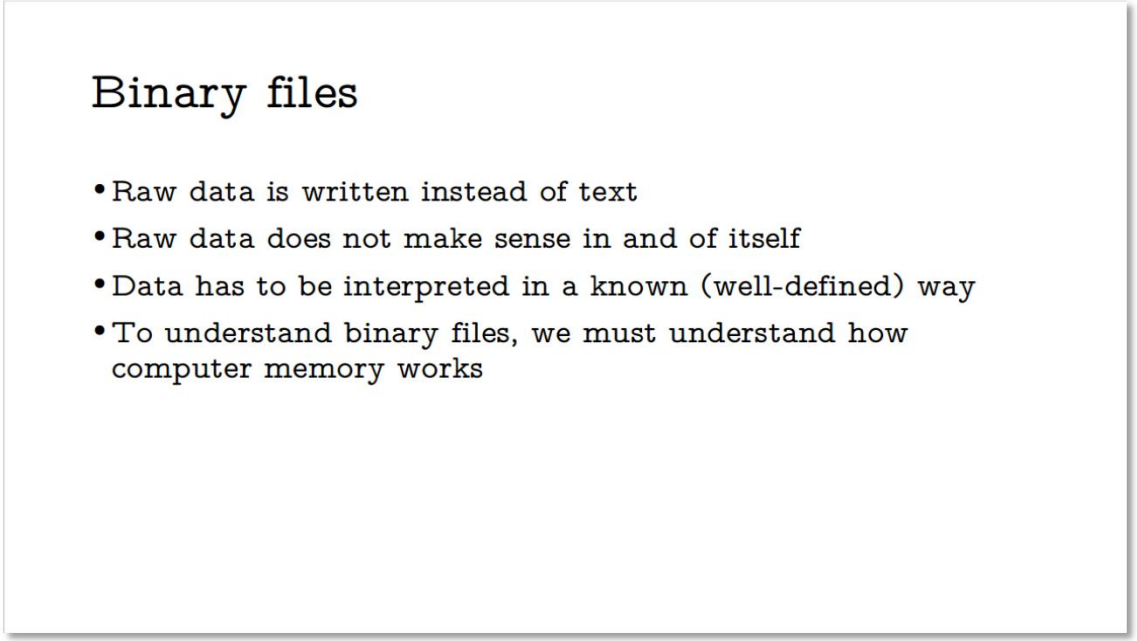
```
david@davidxubuntu:~/temp$ ./binaryRead
Vertex 1
        Position: { -1, 1, 0 }
        Normal: { 0, 0, 1 }
        Tex coords: { 0, 0 }
Vertex 2
        Position: { 1, 1, 0 }
        Normal: { 0, 0, 1 }
        Tex coords: { 1, 0 }
Vertex 3
        Position: { -1, -1, 0 }
        Normal: { 0, 0, 1 }
        Tex coords: { 0, 1 }
Vertex 4
        Position: { 1, -1, 0 }
        Normal: { 0, 0, 1 }
        Tex coords: { 1, 1 }
```

vertices.hexdump

```
   Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 00 00 80 BF 00 00 80 3F 00 00 00 00 00 00 00 00   ...?...?........
00000010: 00 00 00 00 00 00 80 3F 00 00 00 00 00 00 00 00   .......?........
00000020: 00 00 80 3F 00 00 80 3F 00 00 00 00 00 00 00 00   ...?...?........
00000030: 00 00 00 00 00 00 80 3F 00 00 80 3F 00 00 00 00   .......?...?....
00000040: 00 00 80 BF 00 00 80 BF 00 00 00 00 00 00 00 00   ...?...?........
00000050: 00 00 00 00 00 00 80 3F 00 00 00 00 00 00 80 3F   .......?.......?
00000060: 00 00 80 3F 00 00 80 BF 00 00 00 00 00 00 00 00   ...?...?........
00000070: 00 00 00 00 00 00 80 3F 00 00 80 3F 00 00 80 3F   .......?...?...?
```

Binary files in C++ - Reading data

- Exact size of what should be read from the file needs to be known!
- The size of a string, vector, or array isn't always known at compile time
- However, the size of each element is known at compile time




```
// Read entire vertices array from file
in.read(reinterpret_cast<char*>(&vertices), sizeof(Vertex) * VERTEX_COUNT);
```

Binary files in C++ - Reading data (2)

- Solution: split data into headers (fixed size), and data (variable size)
- The size of the header is known at compile time, and it contains how much to read of each variable data type
- Header is designed by you, and documented somewhere
- To read binary file: read the header, and then read data with the size read from the header

```cpp
struct Header
{
    uint32_t firstStringLength;
    uint32_t secondStringLength;
    uint32_t thirdStringLength;
};

int main()
{
    srand(time(nullptr));

    Header header;

    header.firstStringLength = rand() % 16;
    header.secondStringLength = rand() % 16;
    header.thirdStringLength = rand() % 16;

    std::string firstString;
    std::string secondString;
    std::string thirdString;
    for(int i = 0; i < header.firstStringLength; ++i)
        firstString += std::to_string(i);
    for(int i = 0; i < header.secondStringLength; ++i)
        secondString += std::to_string(i);
    for(int i = 0; i < header.thirdStringLength; ++i)
        thirdString += std::to_string(i);

    std::ofstream out("outfile", std::ios::binary);
    // Write header
    out.write((char*)(&header), sizeof(Header));
    // Write variable data
    // ::data() exists for all standard types
    out.write(firstString.data(), header.firstStringLength);
    // ::c_str exists for strings
    out.write(secondString.c_str(), header.secondStringLength);
    // Getting the address of the first element works
    // for vectors and strings
    out.write(&thirdString[0], header.thirdStringLength);
}
```

Binary files in C++ - Reading data (2)

- Solution: split data into headers (fixed size), and data (variable size)
- The size of the header is known at compile time, and it contains how much to read of each variable data type
- Header is designed by you, and documented somewhere
- To read binary file: read the header, and then read data with the size read from the header
- Data of unknown size can be written to file and then read back

```
7  struct Header
8  {
9      uint32_t firstStringLength;
10     uint32_t secondStringLength;
11     uint32_t thirdStringLength;
12 };
13
14 int main()
15 {
16     srand(time(nullptr));
17
18     Header header;
19
20     header.firstStringLength = rand() % 16;
21     header.secondStringLength = rand() % 16;
22     header.thirdStringLength = rand() % 16;
23
24     std::string firstString;
25     std::string secondString;
26     std::string thirdString;
27     for(int i = 0; i < header.firstStringLength; ++i)
28         firstString += std::to_string(i);
29     for(int i = 0; i < header.secondStringLength; ++i)
30         secondString += std::to_string(i);
31     for(int i = 0; i < header.thirdStringLength; ++i)
32         thirdString += std::to_string(i);
33
34     std::ofstream out("outfile", std::ios::binary);
35     // Write header
36     out.write((char*)(&header), sizeof(Header));
37     // Write variable data
38     // ::data() exists for all standard types
39     out.write(firstString.data(), header.firstStringLength);
40     // ::c_str exists for strings
41     out.write(secondString.c_str(), header.secondStringLength);
42     // Getting the address of the first element works
43     // for vectors and strings
44     out.write(&thirdString[0], header.thirdStringLength);
45 }
```

Binary files in C++ - Reading data (2)

- Solution: split data into headers (fixed size), and data (variable size)
- The size of the header is known at compile time, and it contains how much to read of each variable data type
- Header is designed by you, and documented somewhere
- To read binary file: read the header, and then read data with the size read from the header
- Data of unknown size can be written to file and then read back
- File data layout example:


```cpp
struct Header
{
    uint32_t firstStringLength;
    uint32_t secondStringLength;
    uint32_t thirdStringLength;
};

int main()
{
    srand(time(nullptr));

    Header header;

    header.firstStringLength = rand() % 16;
    header.secondStringLength = rand() % 16;
    header.thirdStringLength = rand() % 16;

    std::string firstString;
    std::string secondString;
    std::string thirdString;
    for(int i = 0; i < header.firstStringLength; ++i)
        firstString += std::to_string(i);
    for(int i = 0; i < header.secondStringLength; ++i)
        secondString += std::to_string(i);
    for(int i = 0; i < header.thirdStringLength; ++i)
        thirdString += std::to_string(i);

    std::ofstream out("outfile", std::ios::binary);
    // Write header
    out.write((char*)(&header), sizeof(Header));
    // Write variable data
    // ::data() exists for all standard types
    out.write(firstString.data(), header.firstStringLength);
    // ::c_str exists for strings
    out.write(secondString.c_str(), header.secondStringLength);
    // Getting the address of the first element works
    // for vectors and strings
    out.write(&thirdString[0], header.thirdStringLength);
}
```

Binary files in C++ - Reading data (3)

- Size of header is known at compile time and can be read easily
- Then, according to the specification (made by you), there are 3 strings
- Same principles apply for vectors and arrays, though make sure to read all the data!
  - Don't forget sizeof(dataType)!

```
in.read(&someVector[0], sizeof(someVector[0]) * header.someVectorLength);
in.read(&intVector[0], sizeof(int) * header.intVectorLength);
```

```cpp
int main()
{
    std::ifstream in("outfile", std::ios::binary);

    Header header;
    // Read entire header (fast)
    in.read((char*)&header, sizeof(Header));

    PrintHeader(header);

    // Allocate string with sufficient data
    // IMPORTANT: do not forget to allocate data when
    // working with standard library container (strings, vectors)
    std::string firstString(header.firstStringLength, '\0');
    std::string secondString(header.secondStringLength, '\0');

    // Data can also be allocated with resize
    // IMPORTANT: reserve does _not_ work for this purpose
    std::string thirdString;
    thirdString.resize(header.thirdStringLength, '\0');

    // Read all data at once (fast)
    in.read(&firstString[0], header.firstStringLength);
    in.read(&secondString[0], header.secondStringLength);
    in.read(&thirdString[0], header.thirdStringLength);

    PrintStrings(firstString, secondString, thirdString);
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
david@davidxubuntu:~/temp$ ./headers
header.firstStringLength = 10
header.secondStringLength = 0
header.thirdStringLength = 6
firstString = 0123456789
secondString =
thirdString = 012345
david@davidxubuntu:~/temp$
```

Questions?

Thursday 6/4

- Seminar
- Come up with a draft of your file specification
- Short presentation
  - 5 - 10 minutes per group
  - **Simple** powerpoint or drawing