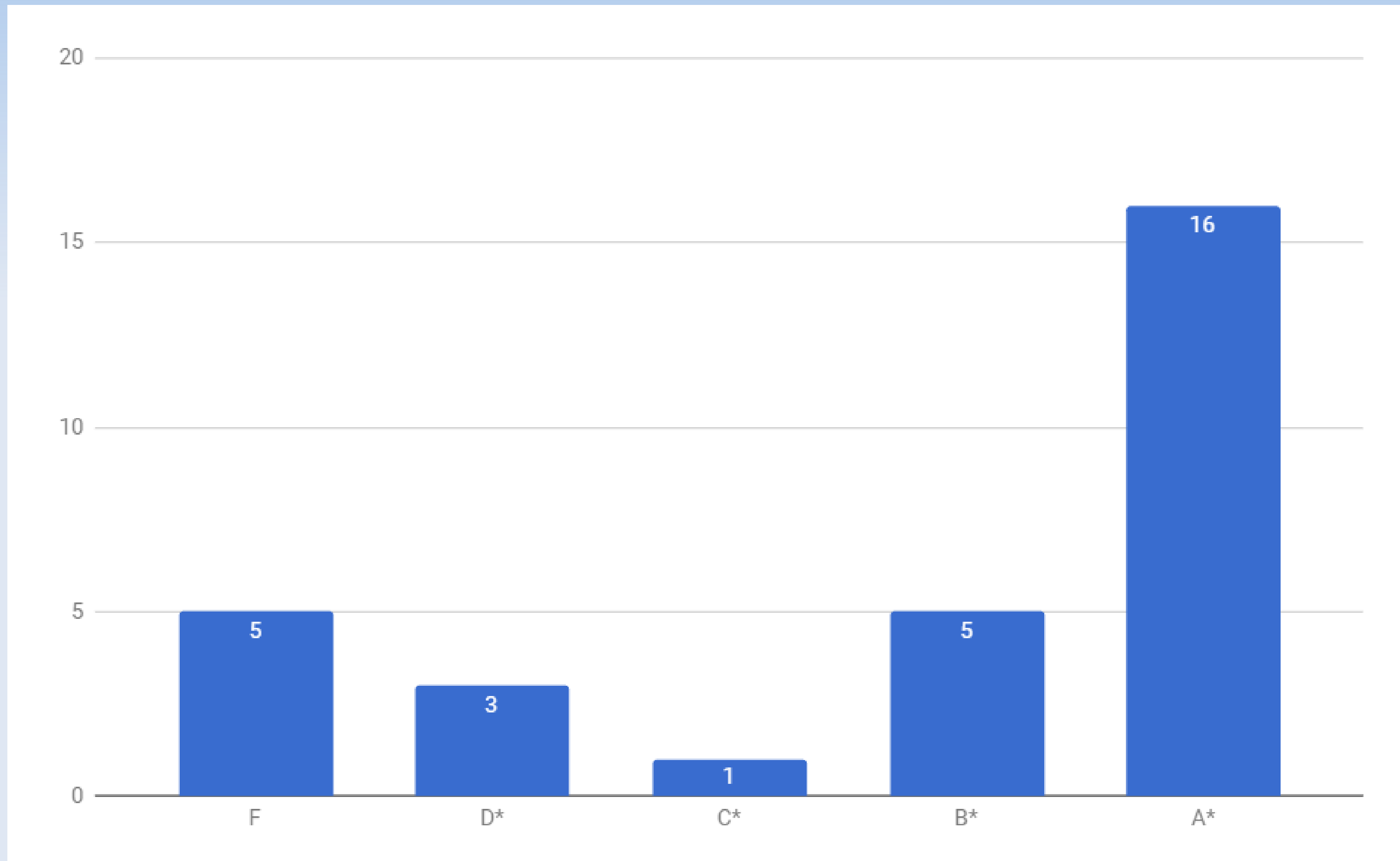
CS 134

Generic Serialization

Grade distribution



General Save / Load

- Write code to save / load things is extremely repetitious
  - Write X, Write Y, Write Z
- A computer could write this code, if it knew what fields were in your struct
- Reflection to the rescue!

Reflection

- Write code about types.
- “What is the type of this object?”
- “What are the fields in this type?”
  - And their names, types
- “Create a new object for this type!”
- “Read a file of this type”
- “Write a file for this type”

Reflection in Java

- Type is a Class<?>
  - That's its type – class Class<?>
- obj.getClass() or Class.forName(“MyType”) or Class<MyType>

- Class has an array of Fields (class Field)
  - .getFields() / .getField(“name”)
  - Field has a name and a value
  - get() / set() – both take the object to get from

Reflection in Java

- Normal code:
  - obj.x = obj.x + 1
- Reflection code:
  - Class c = obj.getClass()
    Field fx = c.getField(“x”);
    fx.setInt(obj, fx.getInt(obj) + 1)

Reflection in Java

- Real power is with .getFields(), which gets all public fields
- Increment ALL fields (assumed to be ints) by 1

```
Class c = obj.getClass();
for( Field fx : c.getFields() ) {
    fx.setInt( obj, fx.getInt( obj ) + 1 )
}
```

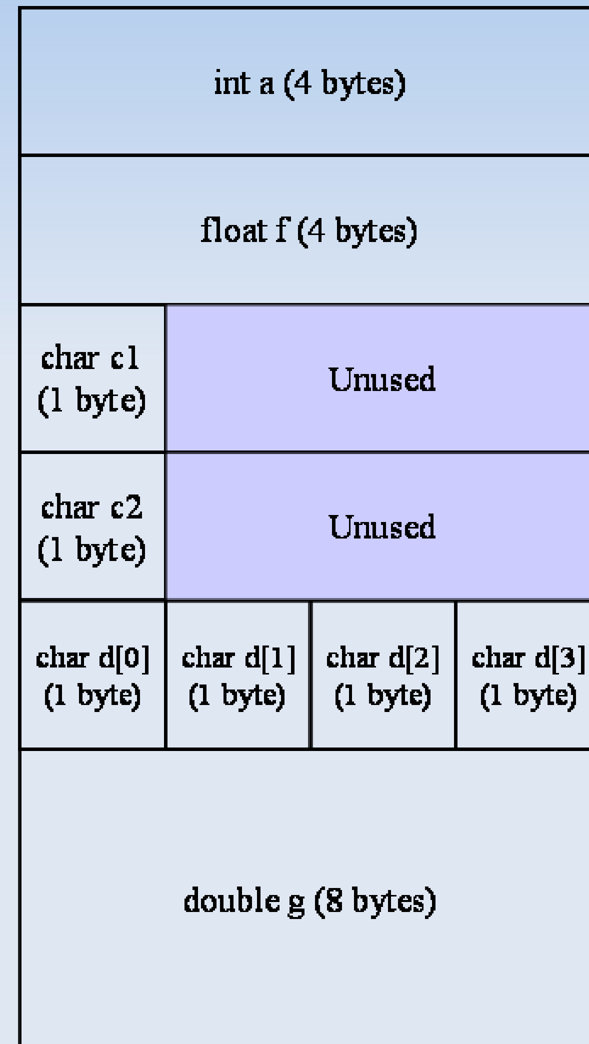
- .getDeclaredFields() also gets private fields

Reflection in Java

Similarly, you could write out (or read in) all the fields of a type via recursion!

Reflection in C

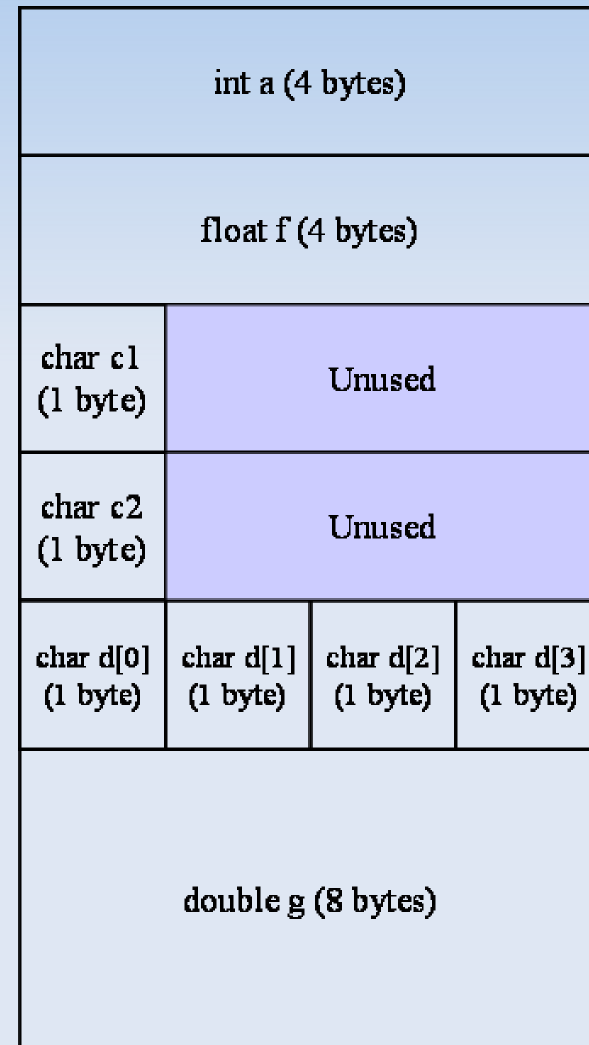
- Unlike Java, C does not automatically create these structures for you
- We will need to create our own.



Reflection in C

- Remember, everything in C is just collection of bytes
- struct MyStruct {

```
struct MyStruct {
    int a;
    float f;
    char c1;
    char c2
    char d[4];
    double g;
};
```



Reflection in C

- Data for each field:
  - Name of field
  - Type of field

int a (4 bytes)

float f (4 bytes)

char c1 (1 byte) | Unused

char c2 (1 byte) | Unused

char d[0] (1 byte) | char d[1] (1 byte) | char d[2] (1 byte) | char d[3] (1 byte)

double g (8 bytes)

Reflection in C

- Data for each field:
  - Name of field
  - Type of field
  - **Storage of field**

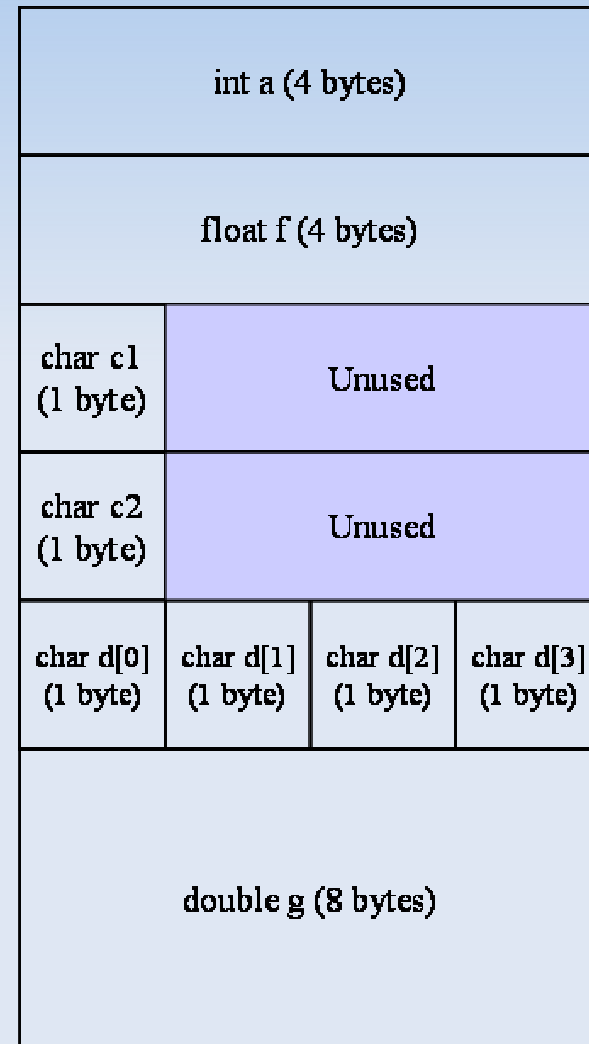
<table>
<tr><td colspan="4">int a (4 bytes)</td></tr>
<tr><td colspan="4">float f (4 bytes)</td></tr>
<tr><td>char c1
(1 byte)</td><td colspan="3">Unused</td></tr>
<tr><td>char c2
(1 byte)</td><td colspan="3">Unused</td></tr>
<tr><td>char d[0]
(1 byte)</td><td>char d[1]
(1 byte)</td><td>char d[2]
(1 byte)</td><td>char d[3]
(1 byte)</td></tr>
<tr><td colspan="4">double g (8 bytes)</td></tr>
</table>

Reflection in C

- Data for each field:
  - Name of field
  - Type of field
  - Storage of field
  - **Memory Offset of field**
- offsetof will help here

offsetof(MyStruct, f) =

offsetof(MyStruct, g) =



Reflection in C

```
struct Type {
    const char* name;
    int size;
    vector<Field> fields;
};

struct Field {
    const char* name;
    Type* type;
    FieldStorage storage;
    int offset;
};
```

```
enum FieldStorage {
    STORAGE_DIRECT,
    STORAGE_DIRECT_PTR,
    STORAGE_VECTOR,
    STORAGE_VECTOR_PTR,
};
```

Reflection in C

```
// What would this struct look
// like?
struct ActorDef {
    const char* name;
    int xPos;
    int yPos;
    const char* fsmDef;
    HealthDef* def;
};
```

```
// Remember, C structs are laid
// out sequentially
```

Reflection in C

```
// What would this struct look
// like?
struct ActorDef {
    const char* name;
    int xPos;
    int yPos;
    const char* fsmDef;
    HealthDef* def;
};
```

```
// Remember, C structs are laid
// out sequentially
Type t = {
    “ActorDef”, 20,
    {
        {
            “name”, &StringType,
            STORAGE_DIRECT,
            0
        },
        {
            “xPos”, &IntType,
            STORAGE_DIRECT,
            4
        },
```

Reflection in C

```
// What would this struct look
// like?
struct ActorDef {
    const char* name;
    int xPos;
    int yPos;
    const char* fsmDef;
    HealthDef* def;
};
```

```
// Remember, C structs are laid
// out sequentially
Type t = {
    "ActorDef", sizeof( ActorDef ),
    {
        {
            "name", &StringType,
            STORAGE_DIRECT,
            offsetof(ActorDef, name)
        },
        {
            "xPos", &IntType,
            STORAGE_DIRECT,
            offsetof(ActorDef, xPos)
        },
```

Reflection in C

```
// Remember, C structs are laid
// out sequentially
Type t = {
    "ActorDef", sizeof( ActorDef ),
    {
        { "name", &StringType, STORAGE_DIRECT,
          offsetof(ActorDef, name) },
         { "xPos", &IntType, STORAGE_DIRECT,
           offsetof(ActorDef, xPos)},
         { "yPos", &IntType, STORAGE_DIRECT,
           offsetof(ActorDef, yPos)},
         { "fsmDef", &StringType, STORAGE_DIRECT,
           offsetof(ActorDef, fsmDef)},
         { "def", &HealthDefType, STORAGE_POINTER,
           offsetof(ActorDef, def)}
    }
}
```

Reflection in C – Get / Set

- Given an offset, how do you get to the memory location for a field:
- (char*)obj + field.offset

- To set an int to 1:
- *(int*)((char*)obj + field.offset) = 1

- To get the value of an int:
- *(int*)((char*)obj + field.offset)

Reflection in C – Get / Set

- Direct:
  - (type*)((char*)obj + field.offset)
- Direct Pointer:
  - (type**)((char*)obj + field.offset)
- Vector:
  - (vector<type>*)((char*)obj + field.offset)
- Vector of Pointers:
  - (vector<type*>*)((char*)obj + field.offset)

Reflection in C – Writing Files

- Write a single generic function that writes out an object of a specified type in JSON:

```
void WriteType( FILE* f, Type* t, void* o )
{
    fprintf( f, "{\n" );
    for( auto field : t.fields ) {
        if( field.type == &IntType || field.type == &FloatType || ... ) {
            WriteBuiltinField( f, t, field, o );
        } else if( field.storage == STORAGE_DIRECT ) {
            fprintf( f, "\t\"%s\": ", field.name );
            WriteType( f, field.type, (char*)obj + field.offset );
        } else if( ... ) {
        }
    }
}
```

Reflection – Final Hookup

- Inspired by Windows!
- Each extension maps to a type
  - .lvl → LevelDefType, .anim → AnimType
- Use opendir() / FindFirstFile() to enumerate all files in your data directory
- Based on extension, load files into a hash table
  - Key is field name “name”
  - Value is full object

Reflection

Questions?

Reflection

Questions?

Final Project

- Due by the day of final (May 21st)
  - But get it done earlier so you can study

- Make a game! A full game.
- You will be graded on three things:
  - Stability – Few bugs, no crashes
  - Completeness – Does it feel like a full game?
  - Fun – The important part of any game

Final Project

- At the very least, every game should have the following:
  - Keyboard controls
  - Title Screen & Game Over screen
  - One level
- Because there are no tests, this is a significant part of your grade. Don't procrastinate!

Final Project

- You can make the game solo or with one other person.

- Suggested timeline for making the game:
  - End of this week: Have a clear design
  - May 4th: Have most code features done
  - May 16th: Have the game done
  - May 17–20: Study for other classes!

Final Project

For the remainder of this class, find a group and figure out exactly what game you want to make.

Before you leave, tell me your team members.