Multiplayer Game Programming

Lecture 1

ITP 484

Who's This Guy?

- Joshua Glazer
- jglazer@usc.edu
- Cofounder and CTO of Naked Sky Entertainment

Who are you

- Names and Expectations

What Will You Learn?

- How to program a real-time, multiplayer game
  - How the Internet works
  - Advantages and disadvantages of different technologies, protocols and topologies
  - How to discuss networking concepts and sound like you know what you're talking about
- What it's like to be a professional game engineer

Syllabus

What Should You Know?

- C++
  - Control structures
  - Local variables
  - Object oriented concepts
  - Pointers
  - Templates
  - References
  - Standard Library

Extremely Basic Competency Check

```
void reverseInPlace( char* inString, int inLength )
{


}
```

Reverse the string in place without dynamically allocating any memory

Shared Pointers

- Referenced counted memory for shared ownership
- Classic Problem:

```
class SampleSprite
{
    SampleSprite( Texture* inTexture ) :
    mTexture( inTexture )
    {
    }

    ~SampleSprite()
    {
     delete mTexture;
    }

    Texture*    mTexture;
};
```

Shared Pointers

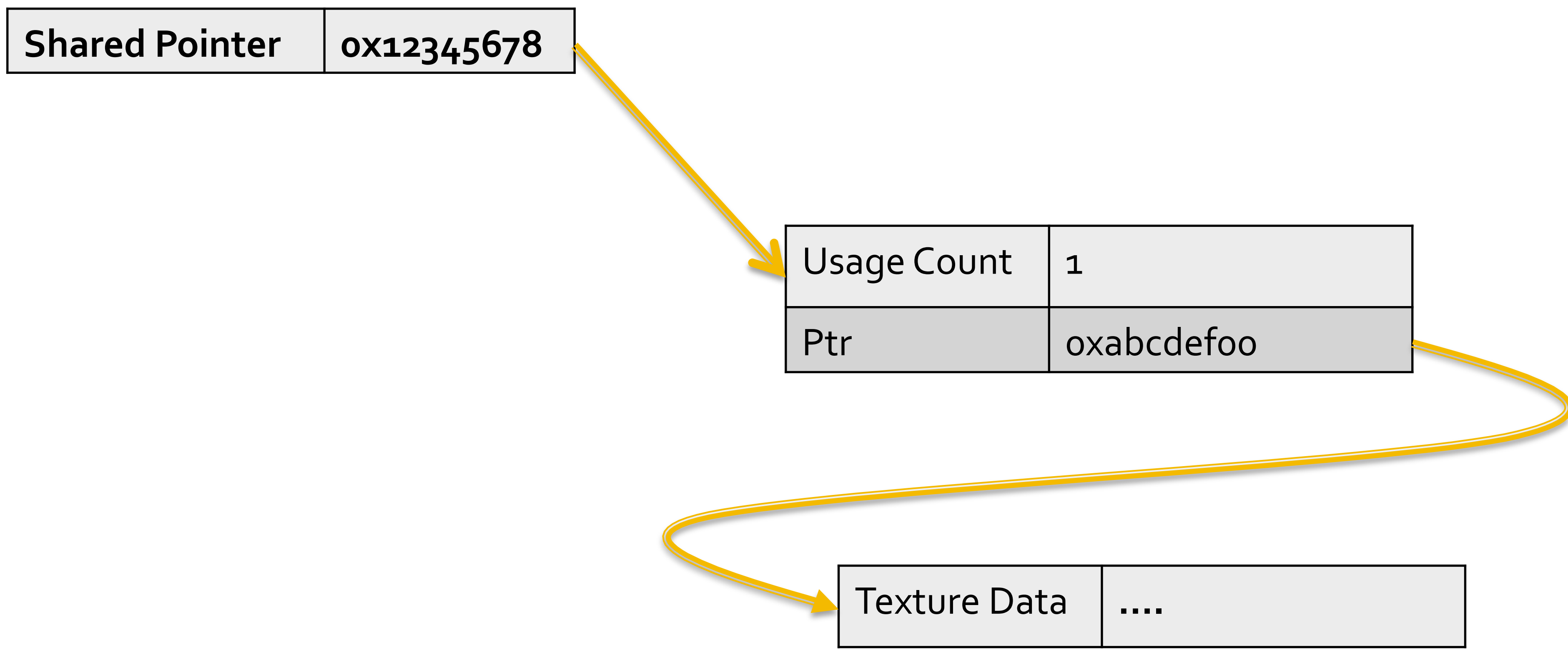
- Automatically track usage count of pointer
- Increment count whenever shared pointer is assigned somewhere
- Decrement count whenever shared pointer is destructed
- When count reaches zero, call delete on pointer

Shared Pointers

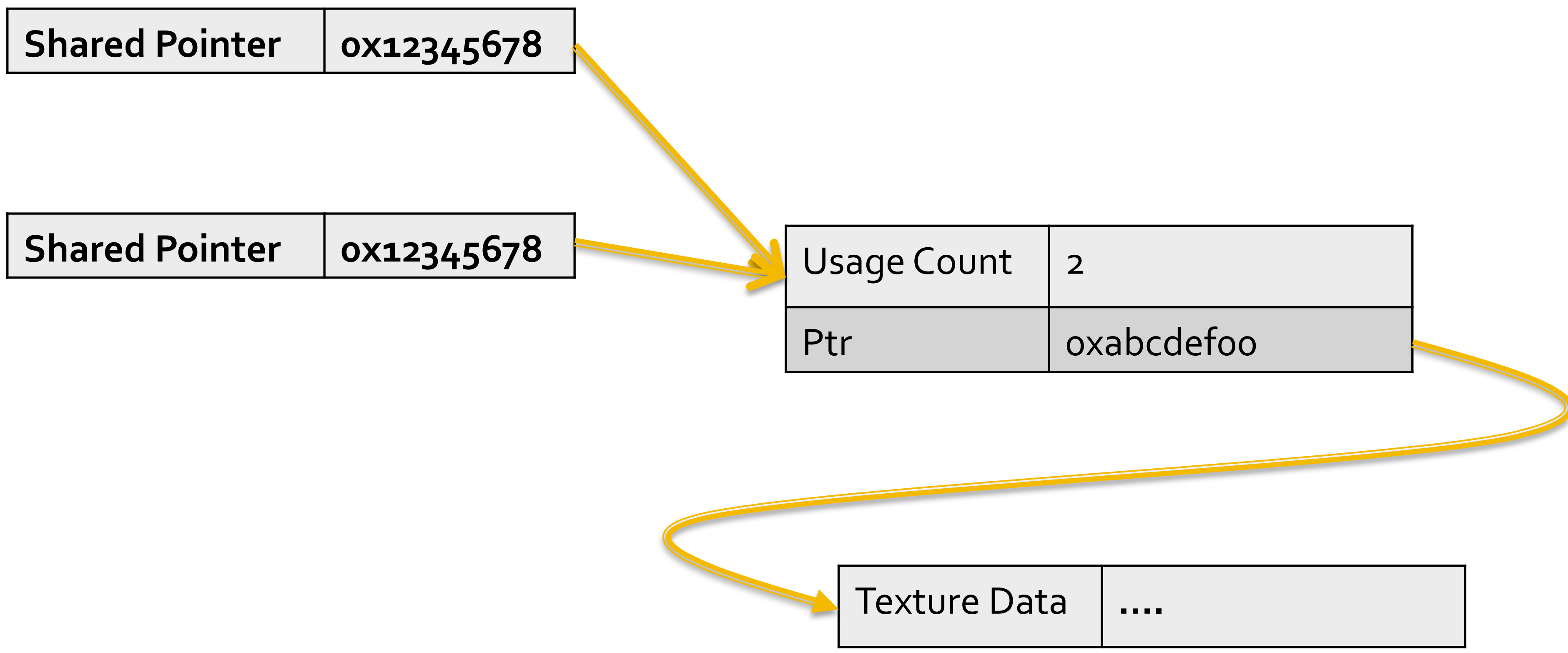
- Better:

```
class SampleSprite
{
    SampleSprite( std::shared_ptr< Texture > inTexture ) :
    mTexture( inTexture )
    {}

    /*
    ~SampleSprite()
    {
      //destructor of shared_ptr called automatically,
      //which deletes the texture if nobody is using it anymore
    }
    */

    std::shared_ptr< Texture >   mTexture;
};
```

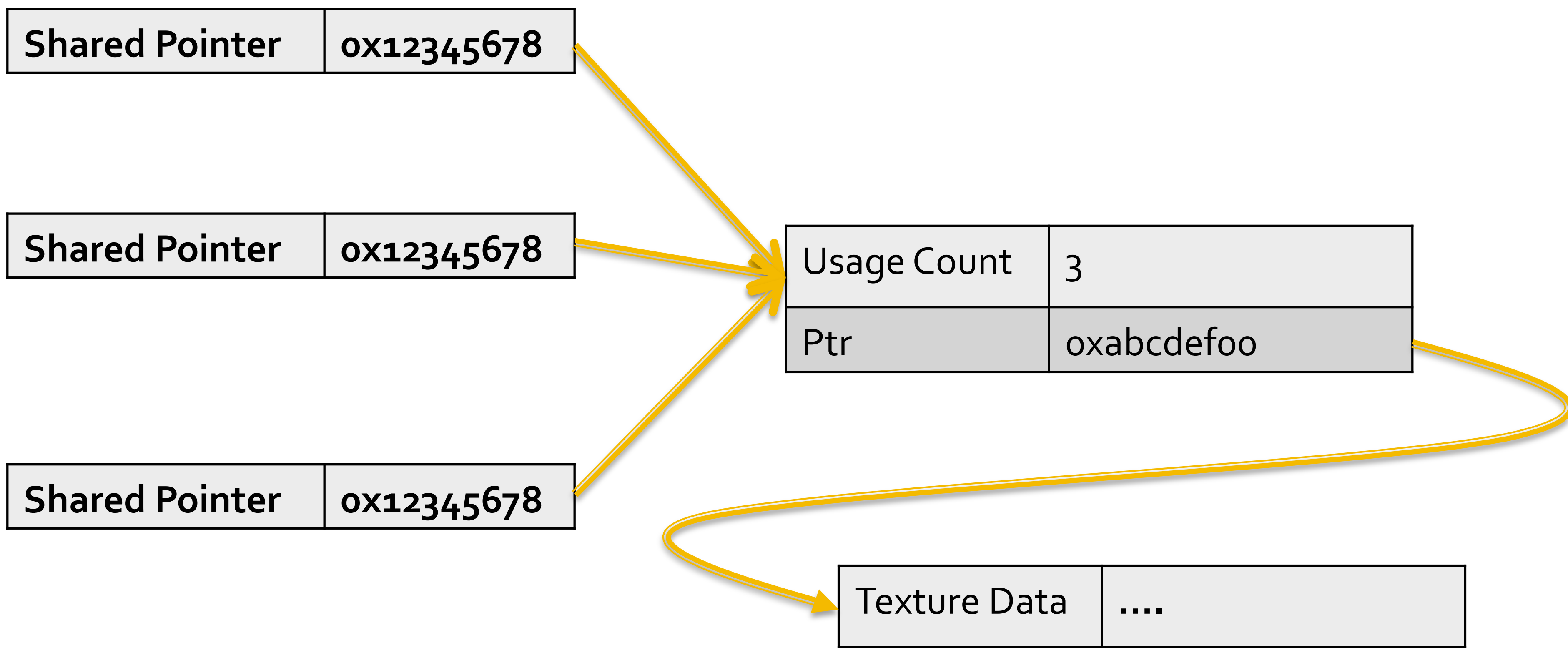
Shared Pointers



Shared Pointers



Shared Pointers



Shared Pointers

Usage Count	0
Ptr	0x00000000

STL Containers

- std::vector
  - dynamically expandable array

```
std::vector< int > numberList;

numberList.push_back( 2 );
numberList.push_back( 3 );
numberList.push_back( 5 );

for( int i = 0; i < numberList.size(); ++i )
{
    printf( "%d\n", numberList[ i ] );
}
```

STL Containers

- std::queue
  - Optimized for insertion at back and removal at front
- std::stack
  - Optimized for insertion at back and removal at back
- std::dequeu
  - Optimized for insertion and removal at either end

Iterators

- Convenient access to STL container elements

```
std::vector< int > numberList;

numberList.push_back( 2 );
numberList.push_back( 3 );
numberList.push_back( 5 );

for( std::vector< int >::iterator it = numberList.begin();
    it != numberList.end();
    ++it )
{
    printf( "%d\n", *it );
}
```

Iterators

- Invalidated when container changed!
- Arithmetic
- Const iterators
- Reverse iterators

```
std::vector< int > numberList;

numberList.push_back( 2 );
numberList.push_back( 3 );
numberList.push_back( 5 );

for( std::vector< int >::reverse_iterator it = numberList.rbegin();
      it != numberList.rend();
      ++it )
{
    printf( "%d\n", *it );
}
```

auto

- Automatic type determination

```
std::vector< int > numberList;

numberList.push_back( 2 );
numberList.push_back( 3 );
numberList.push_back( 5 );

for( auto it = numberList.rbegin(); it != numberList.rend(); ++it )
{
    printf( "%d\n", *it );
}
```

- Still Type Safe
- Don’t overuse- can make code obscure

C++11 For Each

- If container has a begin() and end(), new syntax for iterating through elements!

```
std::vector< int > numberList;

numberList.push_back( 2 );
numberList.push_back( 3 );
numberList.push_back( 5 );

for( auto num: numberList )
{
    printf( "%d\n", num );
}
```

Unicode

- char
  - For 8 bit characters
  - ASCII
  - Can hold multibyte encoded strings
- wchar_t
  - 16 bits on Windows, 32 on Mac / Linux
  - UCS-2 / UCS-4 (subset of UTF16 / UTF32)

STL Strings

- std::string
  - A mutable string of char
  - Constructable from pointer to null terminated array of characters
  - Indexable with []
  - c_str(), length(), replace, substr, resize
  - Benefit over just holding a char* ??
    - Memory management!

Converting Strings

- size_t mbstowcs (wchar_t* dest, const char* src, size_t max);
- size_t wcstombs (char* dest, const wchar_t* src, size_t max);

```
std::string multiByteString( "hi, this is a string" );
wchar_t buffer[ 4096 ];
mbtowc( buffer, multiByteString.c_str(), 4096 );
std::wstring wideString( buffer );
printf( "wideString is %ls", wideString.c_str() );
```

Converting Strings

- More Efficient / Safe:

```
std::string multiByteString( "hi, this is a string" );
std::wstring wideString( multiByteString.size(), '\0' );

size_t convertedLength = mbstowcs( &wideString[ 0 ],
                                   multiByteString.c_str(),
                                   multiByteString.size() );
wideString.resize( convertedLength );
```

Reference

- Semantics of a pointer, without the syntax

```
int a = 3, b = 4;
int c = a;
c = 4;
printf( "a + b = %d", a + b );

int a = 3, b = 4;
int &c = a;
c = 4;
printf( "a + b = %d", a + b );

int a = 3, b = 4;
int *c = &a;
*c = 4;
printf( "a + b = %d", a + b );
```

Const correctness

- The const keyword allow compiler to enforce immutability

```
const std::string hi( "hi" );
hi[ 0 ] = 'X';
```

- Especially useful in function arguments

```
strlen( const char * inString )
```

Const correctness

- And method declarations

```
class Foo
{
    int mMember;

    void Change()
    {
      mMember = 10;
    }

    void Print() const
    {
      printf( "%d", mMember );
    }

};
```

Passing by Const Reference

- Why is this function a sure way to fail a job interview?

```
void printVector( std::vector< int > inVector )
{
    for( auto num: inVector )
    {
     printf( "%d", num );
    }
}
```

- Always pass arguments greater than 4 bytes by const ref!

Online Reference

- http://www.cplusplus.com/
  - Searchable STL reference
- http://www.parashift.com/c++-faq/
  - Everything you always wanted to know about C++ but were afraid to ask