Functors, Lambdas, the **std::function** object, and Assignment 2

Michael Kazhdan

(601.457/657)

Outline

- Recall
- Function passing in C++
- Functors
- Lambdas
- The `std::function` object
- Assignment 2

Recall

We often need to support similar functionality for different types.

compare.cpp

```
#include <iostream>

bool my_comparator_char( const char & t1 , const char & t2 )
{
	return t1<t2;
}

bool my_comparator_int( const int & t1 , const int & t2 )
{
	return t1<t2;
}

bool my_comparator_float( const float & t1 , const float & t2 )
{
	return t1<t2;
}

int main( void )
{
	char c1 = 1 , c2 = 2;
	int i1 =1 , i2 = 2;
	float f1 = 1.f , f2 = 2.f;
	std::cout << my_comparator_char( c1 , c2 ) << std::endl;
	std::cout << my_comparator_int( i1 , i2 )  << std::endl;
	std::cout << my_comparator_float( f1 , f2 )  << std::endl;

	return 0;
}
```

```
>> ./compare
1
1
1
>>
```

Recall

We often need to support similar functionality for different types.

C++ supports overloading so we can give the functions the same name and have C++ figure out which to invoke based on the argument type.

compare.cpp

```
#include <iostream>

bool my_comparator( const char & t1 , const char & t2 )
{
	return t1<t2;
}

bool my_comparator( const int & t1 , const int & t2 )
{
	return t1<t2;
}

bool my_comparator( const float & t1 , const float & t2 )
{
	return t1<t2;
}

int main( void )
{
	char c1 = 1 , c2 = 2;
	int i1 =1 , i2 = 2;
	float f1 = 1.f , f2 = 2.f;
	std::cout << my_comparator( c1 , c2 ) << std::endl;
	std::cout << my_comparator( i1 , i2 )  << std::endl;
	std::cout << my_comparator( f1 , f2 )  << std::endl;

	return 0;
}
```

```
>> ./compare
1
1
1
>>
```

Recall

We often need to support similar functionality for different types.

C++ allows us to handle this more cleanly with templates.

These are recipes that are described using a template parameter that is replaced when the function is invoked (or the object is instantiated).

compare.cpp

```
#include <iostream>

template< typename T >
bool my_comparator( const T & t1 , const T & t2 )
{
        return t1<t2;
}

int main( void )
{
        char c1 = 1 , c2 = 2;
        int i1 =1 , i2 = 2;
        float f1 = 1.f , f2 = 2.f;
        std::cout << my_comparator< char >( c1 , c2 ) << std::endl;
        std::cout << my_comparator< int >( i1 , i2 )  << std::endl;
        std::cout << my_comparator< float >( f1 , f2 )  << std::endl;

        return 0;
}
```

Recall

We often need to support similar functionality for different types.

C++ allows us to handle this more cleanly with templates.

compare.cpp

```
#include <iostream>

template< typename T >
bool my_comparator( const T & t1 , const T & t2 )
{
        return t1<t2;
}

int main( void )
{
        char c1 = 1 , c2 = 2;
        int i1 =1 , i2 = 2;
        float f1 = 1.f , f2 = 2.f;
        std::cout << my_comparator( c1 , c2 ) << std::endl;
        std::cout << my_comparator( i1 , i2 ) << std::endl;
        std::cout << my_comparator( f1 , f2 ) << std::endl;

        return 0;
}
```

These are recipes that are described using a template parameter that is replaced when the function is invoked (or the object is instantiated).

When the parameter type can be derived from the arguments, it does not need to be provided explicitly.

Recall

We often need to support similar functionality for different types.

C++ allows us to handle this more cleanly with templates.

compare.cpp

```
#include <iostream>

template< typename T >
bool my_comparator( const T & t1 , const T & t2 )
{
        return t1<t2;
}

int main( void )
{
        char c1 = 1 , c2 = 2;
        int i1 =1 , i2 = 2;
        float f1 = 1.f , f2 = 2.f;
        std::cout << my_comparator( c1 , c2 ) << std::endl;
        std::cout << my_comparator( i1 , i2 ) << std::endl;
        std::cout << my_comparator( f1 , f2 ) << std::endl;

        return 0;
}
```

These are recipes that are described using a template parameter that is replaced when the function is invoked (or the object is instantiated).

When the parameter type can be derived from the arguments, it does

Note:
The template parameter can be a type or an integer (constant).

Outline

- Recall
- Function passing in C++
- Functors
- Lambdas
- The `std::function` object
- Assignment 2

Function Passing in C

Consider the following C++ code for finding the smallest element in an array.

find_first.cpp

```
#include <iostream>

template< typename T >
bool my_comparator( const T & t1 , const T &t 2 )
{
        return t1<t2;
}

template< typename T , typename T_cmp >
unsigned int find_first( const T * v , size_t count , T_cmp cmp )
{
        unsigned int first = 0;
        for( unsigned int i=1 ; i<count ; i++ ) if( cmp( v[i] , v[first] ) ) first = i;
        return first;
}

int main( void )
{
        int v[12];
        size_t sz = sizeof(v)/sizeof(int);
        for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
        for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;


        unsigned int idx = find_first( v , sz , my_comparator< int > );
        std::cout << idx << " -> " << v[idx] ; std::cout << std::endl;

        return 0;
}
```

```
>> ./find_first
 83 86 77 15 93 35 86 92 49 21 62 27
3 -> 15
>>
```

Function Passing in C

Consider the following C++ code for finding the smallest element in an array.

1. The functions creates (and prints) an array of 12 random values.

find_first.cpp

```
#include <iostream>

template< typename T >
bool my_comparator( const T & t1 , const T & t2 )
{
	return t1<t2;
}

template< typename T , typename T_cmp >
unsigned int find_first( const T * v , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( v[i] , v[first] ) ) first = i;
	return first;
}

int main( void )
{
	int v[12];
	size_t sz = sizeof(v)/sizeof(int);
	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;


	unsigned int idx = find_first( v , sz , my_comparator< int > );
	std::cout << idx << " -> " << v[idx] ; std::cout << std::endl;

	return 0;
}
```

```
>> ./find_first
 83 86 77 15 93 35 86 92 49 21 62 27
```

Function Passing in C

Consider the following C++ code for finding the smallest element in an array.

1. The functions creates (and prints) an array of 12 random values.
2. The index of the smallest value is found (and printed).

find_first.cpp

```
#include <iostream>

template< typename T >
bool my_comparator( const T & t1 , const T & t2 )
{
	return t1<t2;
}

template< typename T , typename T_cmp >
unsigned int find_first( const T * v , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( v[i] , v[first] ) ) first = i;
	return first;
}

int main( void )
{
	int v[12];
	size_t sz = sizeof(v)/sizeof(int);
	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;


	unsigned int idx = find_first( v , sz , my_comparator< int > );
	std::cout << idx << " -> " << v[idx] ; std::cout << std::endl;

	return 0;
}
```

```
>> ./find_first
 83 86 77 15 93 35 86 92 49 21 62 27
3 -> 15
```

Function Passing in C

Consider the following C++ code for finding the smallest element in an array.

1. The functions creates (and prints) an array of 12 random values.
2. The index of the smallest value is found (and printed).

   This is done by setting the index to the first, iterating through the array, and updating if a smaller value is found

find_first.cpp

```
#include <iostream>

template< typename T >
bool my_comparator( const T & t1 , const T & t2 )
{
	return t1<t2;
}

template< typename T , typename T_cmp >
unsigned int find_first( const T * v , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( v[i] , v[first] ) ) first = i;
	return first;
}

int main( void )
{
	int v[12];
	size_t sz = sizeof(v)/sizeof(int);
	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;


	unsigned int idx = find_first( v , sz , my_comparator< int > );
	std::cout << idx << " -> " << v[idx] ; std::cout << std::endl;

	return 0;
}
```

```
>> ./find_first
 83 86 77 15 93 35 86 92 49 21 62 27
3 -> 15
```

Function Passing in C

To find the smallest we need a comparator function taking in two elements and returning a **bool** indicating if the first is smaller than the second.

find_first.cpp

```
#include <iostream>

template< typename T >
bool my_comparator( const T & t1 , const T & t2 )
{
	return t1<t2;
}

template< typename T , typename T_cmp >
unsigned int find_first( const T * v , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( v[i] , v[first] ) ) first = i;
	return first;
}

int main( void )
{
	int v[12];
	size_t sz = sizeof(v)/sizeof(int);
	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;


	unsigned int idx = find_first( v , sz , my_comparator< int > );
	std::cout << idx << " -> " << v[idx] ; std::cout << std::endl;

	return 0;
}
```

```
>> ./find_first
 83 86 77 15 93 35 86 92 49 21 62 27
3 -> 15
```

Function Passing in C

To find the smallest we need a comparator function taking in two elements and returning a **bool** indicating if the first is smaller than the second.

This function is passed as an argument to the **find_first** function.

find_first.cpp

```
#include <iostream>

template< typename T >
bool my_comparator( const T & t1 , const T & t2 )
{
	return t1<t2;
}

template< typename T , typename T_cmp >
unsigned int find_first( const T * v , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( v[i] , v[first] ) ) first = i;
	return first;
}

int main( void )
{
	int v[12];
	size_t sz = sizeof(v)/sizeof(int);
	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;


	unsigned int idx = find_first( v , sz , my_comparator< int > );
	std::cout << idx << " -> " << v[idx] ; std::cout << std::endl;

	return 0;
}
```

```
>> ./find_first
 83 86 77 15 93 35 86 92 49 21 62 27
3 -> 15
```

Function Passing in C

To find the smallest we need a comparator function taking in two elements and returning a **bool** indicating if the first is smaller than the second.

This function is passed as an argument to the **find_first** function.

find_first.cpp

```
#include <iostream>

template< typename T >
bool my_comparator( const T & t1 , const T & t2 )
{
	return t1<t2;
}

template< typename T , typename T_cmp >
unsigned int find_first( const T * v , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( v[i] , v[first] ) ) first = i;
	return first;
}

int main( void )
{
	int v[12];
	size_t sz = sizeof(v)/sizeof(int);
	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;


	unsigned int idx = find_first( v , sz , my_comparator< int > );
	                                  << " -> " << v[idx] ; std::cout << std::endl;
```

```
>> ./find_first
 83 86 77 15 93 35 86 92 49 21 62 27
3 -> 15
```

Note:

Need to specify the type of comparator passed in, not just the recipe.

Function Passing in C

To find the smallest we need a comparator function taking in two elements and returning a **bool** indicating if the first is smaller than the second.

This function is passed as an argument to the **find_first** function.

We don't have to deal with the headache of knowing its type since the second parameter of **find_first** is templated.

find_first.cpp

```
#include <iostream>

template< typename T >
bool my_comparator( const T & t1 , const T & t2 )
{
	return t1<t2;
}

template< typename T , typename T_cmp >
unsigned int find_first( const T * v , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( v[i] , v[first] ) ) first = i;
	return first;
}

int main( void )
{
	int v[12];
	size_t sz = sizeof(v)/sizeof(int);
	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;


	unsigned int idx = find_first( v , sz , my_comparator< int > );
	std::cout << idx << " -> " << v[idx] ; std::cout << std::endl;

	return 0;
}
```

```
>> ./find_first
 83 86 77 15 93 35 86 92 49 21 62 27
3 -> 15
```

Outline

- Recall
- Function passing in C++
- Functors
- Lambdas
- The `std::function` object
- Assignment 2

Functors

Definition:

A functor, a.k.a. function object, is an object that acts like a function by defining the function call operator – `operator()`

Functors

Definition:

A functor, a.k.a. function ob
defining the function call op

find_first_mod.cpp

```
#include <iostream>

template< typename T >
struct my_comparator
{
	bool operator() ( const T & t1 , const T & t2 ) const { return t1<t2; }
};

template< typename T , typename T_cmp >
unsigned int find_first( T * values , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( values[i] , values[first] ) ) first = i;
	return first;
}

int main( void )
{
	int v[12];
	size_t sz = sizeof(v)/sizeof(int);
	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;

	my_comparator< int > cmp;
	unsigned int idx = find_first( v , sz , cmp );
	std::cout << idx << " -> " << v[idx] << std::endl;

	return 0;
}
```

```
>> ./find_first
83 86 77 15 93 35 86 92 49 21 62 27
3 -> 15
>>
```

Functors

Definition:

A *functor*, a.k.a. *function obj*
defining the function call op

- We wrap the function within an object

find_first_mod.cpp

```
#include <iostream>

template< typename T >
struct my_comparator
{
	bool operator() ( const T & t1 , const T & t2 ) const { return t1<t2; }
};

template< typename T , typename T_cmp >
unsigned int find_first( T * values , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( values[i] , values[first] ) ) first = i;
	return first;
}

int main( void )
{
	int v[12];
	size_t sz = sizeof(v)/sizeof(int);
	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;

	my_comparator< int > cmp;
	unsigned int idx = find_first( v , sz , cmp );
	std::cout << idx << " -> " << v[idx] << std::endl;

	return 0;
}
```

```
>> ./find_first
83 86 77 15 93 35 86 92 49 21 62 27
3 -> 15
>>
```

Functors

Definition:

A *functor*, a.k.a. *function obj*
defining the function call op

- We wrap the function within an object
- The functionality is invoked by treating the object as a function

find_first_mod.cpp

```
#include <iostream>

template< typename T >
struct my_comparator
{
	bool operator() ( const T & t1 , const T & t2 ) const { return t1<t2; }
};

template< typename T , typename T_cmp >
unsigned int find_first( T * values , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( values[i] , values[first] ) ) first = i;
	return first;
}

int main( void )
{
	int v[12];
	size_t sz = sizeof(v)/sizeof(int);
	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;

	my_comparator< int > cmp;
	unsigned int idx = find_first( v , sz , cmp );
	std::cout << idx << " -> " << v[idx] << std::endl;

	return 0;
}
```

```
>> ./find_first
83 86 77 15 93 35 86 92 49 21 62 27
3 -> 15
>>
```

Functors

Definition:

A *functor*, a.k.a. *function ob*
defining the function call op

- We wrap the function within an object
- The functionality is invoked by treating the object as a function
- We pass the object to the **find_first** function.

find_first_mod.cpp

```
#include <iostream>

template< typename T >
struct my_comparator
{
	bool operator() ( const T & t1 , const T & t2 ) const { return t1<t2; }
};

template< typename T , typename T_cmp >
unsigned int find_first( T * values , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( values[i] , values[first] ) ) first = i;
	return first;
}

int main( void )
{
	int v[12];
	size_t sz = sizeof(v)/sizeof(int);
	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;

	my_comparator< int > cmp;
	unsigned int idx = find_first( v , sz , cmp );
	std::cout << idx << " -> " << v[idx] << std::endl;

	return 0;
}
```

```
>> ./find_first
83 86 77 15 93 35 86 92 49 21 62 27
3 -> 15
>>
```

Functors

Q: Why do we need functors?

A: To support parametrized functions.

Example:

Suppose we want to create a function that compares numbers, modulo N, where N is a user defined parameter.

- ✗ The function should only take in the values being compared as arguments but should "know" about N.
- ✓ In C++ we can make N be a member data of the functor class.

Functors

Q: Why do we need functor

A: To support parametrized

Example:

Suppose we want to create
modulo N, where N is a use

- ✖ The function should only ta
  should "know" about N.
- ✓ In C++ we can make N be a

find_first_mod.cpp

```
#include <iostream>

template< typename T >
struct my_comparator
{
	bool operator() ( const T & t1 , const T & t2 ) const { return t1%N<t2%N; }
	unsigned int N;
};

template< typename T , typename T_cmp >
unsigned int find_first( T * values , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( values[i] , values[first] ) ) first = i;
	return first;
}

int main( void )
{
	int v[12];
	size_t sz = sizeof(v)/sizeof(int);
	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;

	my_comparator< int > cmp ; std::cin >> cmp.N ;
	unsigned int idx = find_first( v , sz , cmp );
	std::cout << idx << " -> " << v[idx] << " / " << v[idx]%cmp.N << std::endl;

	return 0;
}
```

```
>> echo 6 | ./find_first
 83 86 77 15 93 35 86 92 49 21 62 27
8 -> 49 / 1
>>
```

Outline

- Recall
- Function passing in C++
- Functors
- Lambdas
- The `std::function` object
- Assignment 2

Lambdas

While support for functors enables more powerful code, creating them is cumbersome:

- The functionality is often simple/concise but needs to be defined outside the scope in which it is used

find_first.cpp

```
#include <iostream>

template< typename T >
struct my_comparator
{
	bool operator() ( const T & t1 , const T & t2 ) const { return t1<t2; }
};

template< typename T , typename T_cmp >
unsigned int find_first( const T * v , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( v[i] , v[first] ) ) first = i;
	return first;
}

int main( void )
{
	int v[12];
	size_t sz = sizeof(v)/sizeof(int);
	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;

	my_comparator< int > cmp;
	unsigned int idx = find_first( v , sz , cmp );
	std::cout << idx << " -> " << v[idx] ; std::cout << std::endl;

	return 0;
}
```

```
>> ./find_first
 83 86 77 15 93 35 86 92 49 21 62 27
3 -> 15
>>
```

Lambdas

While support for functors
enables more powerful co
creating them is cumberso

- The functionality is often
simple/concise but need
to be defined outside the
scope in which it is used

find_first.cpp

```
#include <iostream>

template< typename T , typename T_cmp >
unsigned int find_first( const T * v , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( v[i] , v[first] ) ) first = i;
	return first;
}

int main( void )
{
	int v[12];
	size_t sz = sizeof(v)/sizeof(int);
	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;

	unsigned int idx = find_first( v , sz , []( int i1 , int i2 ){ return i1<i2; } );
	std::cout << idx << " -> " << v[idx] ; std::cout << std::endl;

	return 0;
}
```

```
>> ./find_first
 83 86 77 15 93 35 86 92 49 21 62 27
3 -> 15
>>
```

C++ allows us to define lambdas, a.k.a. anonymous functions – functors that are defined on-the-fly.

Lambdas

```
[]( int i1 , int i2 ){ return i1<i2; }
```

C++ allows us to define lambdas – functors that are defined on-the-fly.

Lambdas

```
[]( int i1 , int i2 ){ return i1<i2; }
```

C++ allows us to define lambdas – functors that are defined on-the-fly.

- The definition of the lambda is preceded by square brackets

Lambdas

```
[]( int i1 , int i2 ){ return i1<i2; }
```

C++ allows us to define lambdas – functors that are defined on-the-fly.

- The definition of the lambda is preceded by square brackets
- The arguments to the lambda are described within the parentheses.

Lambdas

```
[]( int i1 , int i2 ){ return i1<i2; }
```

C++ allows us to define lambdas – functors that are defined on-the-fly.

- The definition of the lambda is preceded by square brackets
- The arguments to the lambda are described within the parentheses.
- The body/functionality of the lambda is described within the braces.

Lambdas

```
[]( int i1 , int i2 ){ return i1<i2; }
```

C++ allows us to define lambdas – functors that are defined on-the-fly.

- The definition of the lambda is preceded by square brackets
- The arguments to the lambda are described within the parentheses.
- The body/functionality of the lambda is described within the braces.
- Generally, the return type is determined by the compiler by considering the `return` statement.

  [WARNING] If there are multiple `return` statements in the body of the functor, they should all return the same type

Lambdas

```
[]( int i1 , int i2 ){ return i1<i2; }
```

The contents within the brackets describe what is captured – which local variables the body of the function has access to, and whether the access is by value or by reference.

Examples:

- An empty list means nothing is captured.
- A comma-separated list enumerates the variables captured
  - With a "&" prefix means "captured by reference"
  - Without a "&" prefix means "captured by value"
- Just a "&" inside the brackets means "all variables are captured by reference"

Lambdas

[](int i

The contents within the b
local variables the body of
access is by value or by re

find_first_mod.cpp

```
#include <iostream>

template< typename T , typename T_cmp >
unsigned int find_first( const T * v , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( v[i] , v[first] ) ) first = i;
	return first;
}

int main( void )
{
	int v[12];
	size_t sz = sizeof(v)/sizeof(int);
	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;

	unsigned int N;
	std::cin >> N;
	unsigned int idx = find_first( v , sz , [N]( int i1 , int i2 ){ return i1%N<i2%N; } );
	std::cout << idx << " -> " << v[idx] ; std::cout << " / " << v[idx]%N << std::endl;

	return 0;
}
```

```
>> echo 6 | ./find_first_mod
 83 86 77 15 93 35 86 92 49 21 62 27
8 -> 49 / 1
>>
```

Examples:

- An empty list means not
- A comma-separated list enumerates the variables captured
  - With a "&" prefix means "captured by reference"
  - Without a "&" prefix means "captured by value"
- Just a "&" inside the brackets means "all variables are captured by reference"

Lambdas

Providing a lambda as an argument expands out to:

1. Having the compiler creates a temporary type defining the function call operator and captured variables.

find_first_mod.cpp

```
#include <iostream>

template< typename T , typename T_cmp >
unsigned int find_first( const T * v , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( v[i] , v[first] ) ) first = i;
	return first;
}

int main( void )
{
	int v[12];
	size_t sz = sizeof(v)/sizeof(int);
	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;

	unsigned int N;
	std::cin >> N;

	struct some_terribly_awful_type_name
	{
		unsigned int N;
		some_terribly_awful_type_name( unsigned int N ) : N(N) {}
		bool operator() ( int i1 , int i2 ){ return i1%N < i2%N; }
	};
	some_terribly_awful_type_name statn(N);
	unsigned int idx = find_first( v , sz , statn );
	std::cout << idx << " -> " << v[idx] ; std::cout << " / " << v[idx]%N << std::endl;

	return 0;
}
```

Lambdas

Providing a lambda as an argument expands out to:

1. Having the compiler creates a temporary type defining the function call operator and captured variables.
2. Declaring an object of that type.

find_first_mod.cpp

```
#include <iostream>

template< typename T , typename T_cmp >
unsigned int find_first( const T * v , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( v[i] , v[first] ) ) first = i;
	return first;
}

int main( void )
{
	int v[12];
	size_t sz = sizeof(v)/sizeof(int);
	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;

	unsigned int N;
	std::cin >> N;

	struct some_terribly_awful_type_name
	{
		unsigned int N;
		some_terribly_awful_type_name( unsigned int N ) : N(N) {}
		bool operator() ( int i1 , int i2 ){ return i1%N < i2%N; }
	};
	some_terribly_awful_type_name statn(N);
	unsigned int idx = find_first( v , sz , statn );
	std::cout << idx << " -> " << v[idx] ; std::cout << " / " << v[idx]%N << std::endl;

	return 0;
}
```

Lambdas

Providing a lambda as an argument expands out to:

1. Having the compiler creates a temporary type defining the function call operator and captured variables.
2. Declaring an object of that type.
3. Passing the object as an argument to the function.

find_first_mod.cpp

```
#include <iostream>

template< typename T , typename T_cmp >
unsigned int find_first( const T * v , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( v[i] , v[first] ) ) first = i;
	return first;
}

int main( void )
{
	int v[12];
	size_t sz = sizeof(v)/sizeof(int);
	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;

	unsigned int N;
	std::cin >> N;

	struct some_terribly_awful_type_name
	{
		unsigned int N;
		some_terribly_awful_type_name( unsigned int N ) : N(N) {}
		bool operator() ( int i1 , int i2 ){ return i1%N < i2%N; }
	};
	some_terribly_awful_type_name statn(N);
	unsigned int idx = find_first( v , sz , statn );
	std::cout << idx << " -> " << v[idx] ; std::cout << " / " << v[idx]%N << std::endl;

	return 0;
}
```

Outline

- Recall
- Function passing in C++
- Functors
- Lambdas
- The `std::function` object
- Assignment 2

The **std::function** Object

```
[]( int i1 , int i2 ){ return i1<i2; }
```

Q: Given the ability to define a lambda, how do we declare one?

- Because the compiler defines it on-the-fly, the type is unspecified.

We may want to explicitly declare a lambda:

- If we want to use the same lambda multiple times
- If the lambda definition takes multiple-lines and we want to separate the definition from the invocation

The **std::function** Object

```
std::function< bool ( int , int ) > sort_lambda
    = []( int i1 , int i2 ){ return i1<i2; };
```

C++ defines a generic function wrapper object, **std::function**, that can be assigned by a lambda (more generally, functor or even function).

The **std::function** Object

```
std::function< bool ( int , int ) > sort_lambda
    = []( int i1 , int i2 ){ return i1<i2; };
```

C++ defines a generic function wrapper object, **std::function**, that can be assigned by a lambda (more generally, functor or even function).

- The first template parameter describes the return type.

The **std::function** Object

```
std::function< bool ( int , int ) > sort_lambda
    = []( int i1 , int i2 ){ return i1<i2; };
```

C++ defines a generic function wrapper object, **std::function**, that can be assigned by a lambda (more generally, functor or even function).

- The first template parameter describes the return type.
  - This must match the type returned by the definition.

The **std::function** Object

```
std::function< bool ( int , int ) > sort_lambda
    = []( int i1 , int i2 ){ return i1<i2; };
```

C++ defines a generic function wrapper object, **std::function**, that can be assigned by a lambda (more generally, functor or even function).

- The first template parameter describes the return type.
- The remaining template parameters are enclosed in parentheses and describe the argument types.

The **std::function** Object

```
std::function< bool ( int , int ) > sort_lambda
    = [](  int i1 , int i2 ){ return i1<i2; };
```

C++ defines a generic function wrapper object, **std::function**, that can be assigned by a lambda (more generally, functor or even function).

- The first template parameter describes the return type.
- The remaining template parameters are enclosed in parentheses and describe the argument types.
  - These must match the argument types in the definition

The std::functi

std::function<
= [](int i

C++ defines a generic funct
be assigned by a lambda (r

- The first template parame
- The remaining template p
  the argument types.

find_first_mod.cpp

```
#include <iostream>
#include <functional>

template< typename T , typename T_cmp >
unsigned int find_first( const T * v , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( v[i] , v[first] ) ) first = i;
	return first;
}

int main( void )
{
	unsigned int idx , N;
	std::cin >> N;
	std::function< bool ( int , int ) > sort_lambda = [N]( int i1 , int i2 ){ return i1%N<i2%N; };

	int v[12];
	size_t sz = sizeof(v)/sizeof(int);

	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;

	idx = find_first( v , sz , sort_lambda );
	std::cout << idx << " -> " << v[idx] << " / " << v[idx]%N << std::endl;

	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << "

	idx = find_first( v , sz , sort_lambda );
	std::cout << idx << " -> " << v[idx] << " / " << v[idx
	return 0;
}
```

```
>> echo 6 | ./find_first_mod
 83 86 77 15 93 35 86 92 49 21 62 27
8 -> 49 / 1
 90 59 63 26 40 26 72 36 11 68 67 29
0 -> 90 / 0
>>
```

The **std::function** Object

```
[]( int i1 , int i2 ){ return i1<i2; }
```

Q: Given the ability to define a lambda, how do we declare one?

- ✗ Because the compiler defines it on-the-fly, the type is unspecified.
- ✓ We don't need to know the type, we just need the compiler to know it.

We may want to explicitly declare a lambda:

- If we want to use the same lambda multiple times
- If the lambda definition takes multiple-lines and we want to separate the definition from the invocation

The **std::function** Object

```
auto sort_lambda = []( int i1 , int i2 ){ return i1<i2; };
```

When C++ knows an object's type, we can also use the keyword **auto** to declare the object.

The std::functi

auto sort_lambda

When C++ knows an objec
to declare the object.

find_first_mod.cpp

```
#include <iostream>


template< typename T , typename T_cmp >
unsigned int find_first( const T *v , size_t count , T_cmp cmp )
{
	unsigned int first = 0;
	for( unsigned int i=1 ; i<count ; i++ ) if( cmp( v[i] , v[first] ) ) first = i;
	return first;
}

int main( void )
{
	unsigned int idx , N;
	std::cin >> N;
	auto sort_lambda = [N]( int i1 , int i2 ){ return i1%N<i2%N; };

	int v[12];
	size_t sz = sizeof(v)/sizeof(int);

	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;

	idx = find_first( v , sz , sort_lambda );
	std::cout << idx << " -> " << v[idx] << " / " << v[idx]%N << std::endl;

	for( unsigned int i=0 ; i<sz ; i++ ) v[i] = rand()%100;
	for( unsigned int i=0 ; i<sz ; i++ ) std::cout << " " << v[i] ; std::cout << std::endl;

	idx = find_first( v , sz , sort_lambda );
	std::cout << idx << " -> " << v[idx] << " / " << v[idx]%N << std::endl;
	return 0;
}
```

The std::function

std::function:

- ✓ Descriptive type:
  Easier to understand what the object should be doing
- ✗ More cumbersome
- ✗ Can be slower

auto:

- ✗ Generic type:
  Harder to understand what the object should be doing
- ✓ Cleaner
- ✓ Can be faster

find_first_mod.cpp

```
#include <iostream>
#include <functional>

template< typename T >
unsigned int find_first( const T * v , size_t count , std::function< bool ( T , T ) > cmp )
{
        unsigned int first = 0;
        for( unsigned int i=1 ; i<count ; i++ ) if( cmp( v[i] , v[first] ) ) first = i;
        return first;
}
...
```

find_first_mod.cpp

```
#include <iostream>


template< typename T , typename T_cmp >
unsigned int find_first( const T * v , size_t count , T_cmp cmp )
{
        unsigned int first = 0;
        for( unsigned int i=1 ; i<count ; i++ ) if( cmp( v[i] , v[first] ) ) first = i;
        return first;
}
...
```

Outline

- Recall
- Function passing in C++
- Functors
- Lambdas
- The `std::function` object
- Assignment 2
  - The `Util::Polynomial` class
  - The `Shape::processFirstIntersection` method

Util::Polynomial< Dim , Degree >

Suppose you would like to define a quadratic polynomial describing a circle with radius 2, centered about the point $(1,2)$ in 2D:

$$P(x,y) = (x-1)^2 + (y-2)^2 - 4$$
$$= x^2 + y^2 - 2x - 4y + 1$$

Util::Polynomial< Dim , Degree >

$$P(x, y) = x^2 + y^2 - 2x - 4y + 1$$

Defining the polynomial:

```
// Declare a polynomial in two variables of degree 2
Util::Polynomial2D< 2 > poly2D;

// Set the (non-zero) coefficients
poly2D.coefficient(2u,0u) = 1;      // the x^2*y^0 term
poly2D.coefficient(0u,2u) = 1;      // the x^0*y^2 term
poly2D.coefficient(1u,0u) = -2;     // the x^1*y^0 term
poly2D.coefficient(0u,1u) = -4;     // the x^0*y^1 term
poly2D.coefficient(0u,0u) = 1;      // the x^0*y^0 term
```

Util::Polynomial< Dim , Degree >

$$P(x, y) = x^2 + y^2 - 2x - 4y + 1$$

Evaluating the polynomial:

```
// The x and y coordinates at which to evaluate P
double x , y;
...
// The point at which you want to evaluate P
Util::Point2D p(x,y);
...
// Evaluate P at the prescribed point
double value1 = poly2D( p );
double value2 = poly2D( x , y );
```

Util::Polynomial< Dim , Degree >

$$P(x, y) = x^2 + y^2 - 2x - 4y + 1$$

Restricting the polynomial:

```
// The ray to which you want to restrict P
Util::Ray2D ray;
...
// The restriction is a polynomial in one variable of degree 2
Util::Polynomial1D< 2 > poly1D = poly2D( ray );
```

Util::Polynomial< Dim , Degree >

$$P(x, y) = x^2 + y^2 - 2x - 4y + 1$$

Find the roots (of a 1D polynomial):

```
Util::Polynomial1D< 2 > poly1D;
...
double roots[2];
unsigned int rootNum = p.roots( roots );
```

Note:
The number of roots may be less than the degree of the polynomial, so check the value of `rootNum`.

Util::Polynomial< Dim , Degree >

$$P(x, y) = x^2 + y^2 - 2x - 4y + 1$$

Differentiating the polynomial:

```
// The partial derivative with respect to x
Util::Polynomial2D< 1 > dx = poly2D.d( 0 );

// The partial derivative with respect to y
Util::Polynomial2D< 1 > dy = poly2D.d( 1 );
```

Note:
When you take a derivative, the degree of the polynomial reduces by one.

Outline

- Function passing
- The Util::Polynomial class
- The Shape::processFirstIntersection method
  - The ShapeList::processFirstIntersection method

Shape::processFirstInt

```
virtual bool processFirstIntersection
(
      const Util::Ray3D & ray ,
      const Util::BoundingBox1D & range ,
      const RayIntersectionFilter & rFilter ,
      const RayIntersectionKernel & rKernel ,
      ShapeProcessingInfo spInfo ,
      unsigned int tIdx
) const = 0;
```

Ray-Casting:

Given a Shape, we want to (1) find the first intersection with a Util::Ray3D and (2) perform some computation using that intersection information.

Shape::processFirstInt

```
virtual bool Shape::processFirstIntersection
(
    const Util::Ray3D & ray ,
    const Util::BoundingBox1D & range ,
    const RayIntersectionFilter & rFilter ,
    const RayIntersectionKernel & rKernel ,
    ShapeProcessingInfo spInfo ,
    unsigned int tIdx
) const = 0;
```

Ray-Casting:

Given a Shape, we want to (1) find the first intersection with a Util::Ray3D and (2) perform some computation using that intersection information.

ray: The Util::Ray3D we are intersecting with, in the local coordinate frame of the Shape.
(The transformation into local coordinates is taken care of for you in AffineShape::processFirstIntersection)

Shape::processFirstInt

```
virtual bool Shape::processFirstIntersection
(
    const Util::Ray3D & ray ,
    const Util::BoundingBox1D & range ,
    const RayIntersectionFilter & rFilter ,
    const RayIntersectionKernel & rKernel ,
    ShapeProcessingInfo spInfo ,
    unsigned int tIdx
) const = 0;
```

Ray-Casting:

Given a Shape, we want to (1) find the first intersection with a Util::Ray3D and (2) perform some computation using that intersection information.

range: The range, $[start, end]$ of values over which we are interested in an intersection. (If the ray intersects the shape at a time t outside the range, we do not consider that as a candidate intersection.)

Example:

- For shadow testing with point/spot lights, we can use this to restrict ourselves to surfaces between the source of the ray and the light.
- Can use this to discard intersections too close to the start of the ray (to make sure we don't keep intersecting the same point).

Shape::processFirstInt

```
virtual bool Shape::processFirstIntersection
(
    const Util::Ray3D & ray ,
    const Util::BoundingBox1D & range ,
    const RayIntersectionFilter & rFilter ,
    const RayIntersectionKernel & rKernel ,
    ShapeProcessingInfo spInfo ,
    unsigned int tIdx
) const = 0;
```

Ray-Casting:

Given a Shape, we want to (1) find the first intersection with a Util::Ray3D and (2) perform some computation using that intersection information.

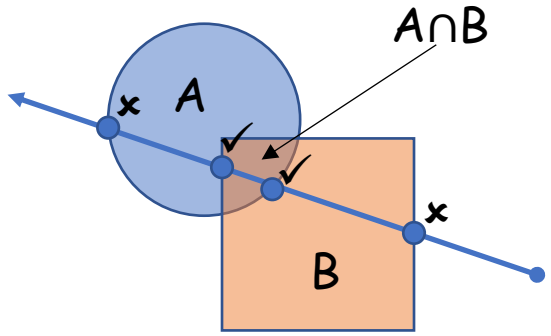
rFilter: An object of type

std::function< bool (double) >

that can be used to, more generally, filter out a candidate intersection as a function of the time t.

Example:

- When considering the intersection of shapes A and B, we could consider:
  - Intersections with A, only if they are inside of B.
  - Intersections with B, only if they are inside of A.



Shape::processFirstInt

```
virtual bool Shape::processFirstIntersection
(
    const Util::Ray3D & ray ,
    const Util::BoundingBox1D & range ,
    const RayIntersectionFilter & rFilter ,
    const RayIntersectionKernel & rKernel ,
    ShapeProcessingInfo spInfo ,
    unsigned int tIdx
) const = 0;
```

Ray-Casting:

Given a Shape, we want to (1) find the first intersection with a Util::Ray3D and (2) perform some computation using that intersection information.

rKernel: A function object performing the computation, of type

```
std::function< void ( const ShapeProcessingInfo & ,
                      const RayShapeIntersectionInfo & ) >
```

that is to be invoked with

- RayShapeIntersectionInfo: information about the first ray-shape intersection, and
- ShapeProcessingInfo: accumulated transformations.

Example:

- This can be used to update/set the color of a pixel.

Shape::processFirstInt

```
virtual bool Shape::processFirstIntersection
(
    const Util::Ray3D & ray ,
    const Util::BoundingBox1D & range ,
    const RayIntersectionFilter & rFilter ,
    const RayIntersectionKernel & rKernel ,
    ShapeProcessingInfo spInfo ,
    unsigned int tIdx
) const = 0;
```

Ray-Casting:

Given a Shape, we want to (1) find the first intersection with a Util::Ray3D and (2) perform some computation using that intersection information.

spInfo: An object storing the global-to-local and local-to-global transformations needed to transform points, directions, and normals, (as well as other state). (The updates of the transformations is taken care of for you in AffineShape::processFirstIntersection)

Shape::processFirstInt

```
virtual bool Shape::processFirstIntersection
(
      const Util::Ray3D & ray ,
      const Util::BoundingBox1D & range ,
      const RayIntersectionFilter & rFilter ,
      const RayIntersectionKernel & rKernel ,
      ShapeProcessingInfo spInfo ,
      unsigned int tIdx
) const = 0;
```

Ray-Casting:

Given a Shape, we want to (1) find the first intersection with a Util::Ray3D and (2) perform some computation using that intersection information.

tIdx: If you are implementing the ray-tracer as a multi-threaded application, this integer gives the index of the thread processing this ray-shape intersection.

Example:

- For efficiency, you may pre-allocate temporary arrays for computation.
  To avoid multiple threads writing to the same array, you can allocate a separate array per thread and use tIdx to determine which array this thread should use.

ShapeList::processFirstIntersection

Challenge:

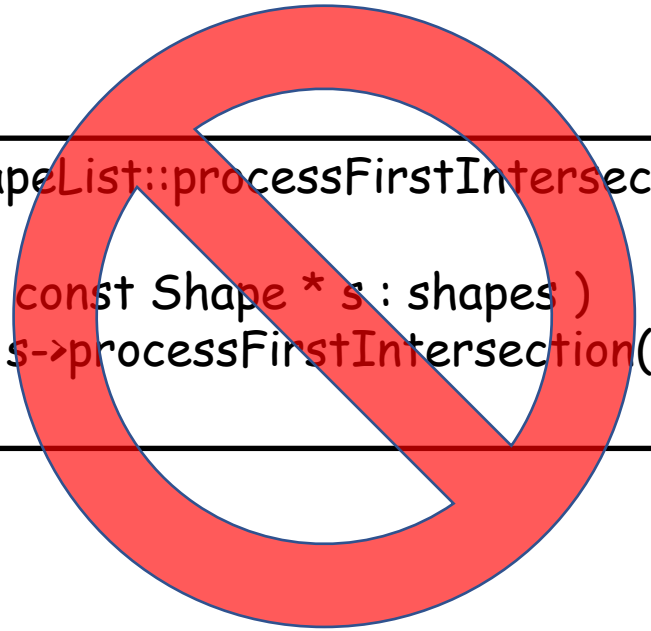
Invoking **rKernel** on the first intersection with a **ShapeList** is not the same as invoking **rKernel** on the first intersection with each of the **ShapeList**'s children.

Example:

If **rKernel** sets the pixel color, we would be setting the color with the first intersection of the last intersected shape, not the first intersection.

```
class ShapeList : public Shape
{
    ...
    std::vector< Shape * > shapes;
    ...
};
```

```
bool ShapeList::processFirstIntersection( ... )
{
    for( const Shape * s : shapes )
        s->processFirstIntersection( ... );
};
```



ShapeList::processFirstIntersection

Solution:

- Create a local variable to track the information about the actual first intersection (iInfo)

```
class ShapeList : public Shape
{
      …
      std::vector< Shape * > shapes;
      …
};
```

```
bool ShapeList::processFirstIntersection( … )
{
      RayShapeIntersectionInfo iInfo;
      RayIntersectionKernel k = [&iInfo]( … )
      {
            …
      };
      for( const Shape * s : shapes )
            s->processFirstIntersection( … , k , … );
      rKernel( … , iInfo );
};
```

ShapeList::processFirstIntersection

Solution:

- Create a local variable to track the information about the actual first intersection (iInfo)
- Define a new kernel to set the iInfo variable if a closer intersection was found (kernel)

```
class ShapeList : public Shape
{
    ...
    std::vector< Shape * > shapes;
    ...
};
```

```
bool ShapeList::processFirstIntersection( ... )
{
    RayShapeIntersectionInfo iInfo;
    RayIntersectionKernel k = [&iInfo]( ... )
    {
        ...
    };
    for( const Shape * s : shapes )
        s->processFirstIntersection( ... , k , ... );
    rKernel( ... , iInfo );
};
```

ShapeList::processFirstIntersection

Solution:

- Create a local variable to track the information about the actual first intersection (iInfo)
- Define a new kernel to set the iInfo variable if a closer intersection was found (kernel)
- Process the first intersections of the ray with the children using the new kernel

```
class ShapeList : public Shape
{
    ...
    std::vector< Shape * > shapes;
    ...
};
```

```
bool ShapeList::processFirstIntersection( ... )
{
    RayShapeIntersectionInfo iInfo;
    RayIntersectionKernel k = [&iInfo]( ... )
    {
        ...
    };
    for( const Shape * s : shapes )
        s->processFirstIntersection( ... , k , ... );
    rKernel( ... , iInfo );
};
```

ShapeList::processFirstIntersection

Solution:

- Create a local variable to track the information about the actual first intersection (iInfo)
- Define a new kernel to set the iInfo variable if a closer intersection was found (kernel)
- Process the first intersections of the ray with the children using the new kernel
- Invoke the original kernel with the information about the closest intersection

```
class ShapeList : public Shape
{
    ...
    std::vector< Shape * > shapes;
    ...
};
```

```
bool ShapeList::processFirstIntersection( ... )
{
    RayShapeIntersectionInfo iInfo;
    RayIntersectionKernel k = [&iInfo]( ... )
    {
        ...
    };
    for( const Shape * s : shapes )
        s->processFirstIntersection( ... , k , ... );
    rKernel( ... , iInfo );
};
```