601.220 Intermediate Programming

Lambdas and the `auto` keyword

Outline

- Passing Functions in C vs. C++
- Functors
- Lambdas
- The `auto` keyword

Passing Functions in C and C++

In addition to passing data (e.g. `int`s, `float`s, `struct`s, pointers, etc.) as arguments to functions, C and C++ support passing functionality:

- C allows passing *function pointers*
- C++ also allows classes to have member functions, so that passing an object implicitly passes the associated functionality

Passing Functions in

Sorting:

In C, when we want to sort a list of values, we can use the **qsort** function.

sort.c

```
#include <stdio.h>
#include <stdlib.h>
int compare_int( const void *v1 , const void *v2 )
{
	if     ( *(int *)v1<*(int *)v2 ) return -1;
	else if( *(int *)v2<*(int *)v1 ) return  1;
	else                             return  0;
}
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++ ) printf( " %d" , v[i] );
	printf( "\n" );

	qsort( v , sz , sizeof(int) , compare_int );

	for( unsigned int i=0 ; i<sz ; i++ ) printf( " %d" , v[i] );
	printf( "\n" );

	return 0;
}
```

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

Passing Functions in

Sorting:

In C, when we want to sort a list of values, we can use the **qsort** function.

sort.c

```
#include <stdio.h>
#include <stdlib.h>
int compare_int( const void *v1 , const void *v2 )
{
        if      ( *(int *)v1<*(int *)v2 ) return -1;
        else if( *(int *)v2<*(int *)v1 ) return  1;
        else                              return  0;
}
int main( void )
{
        ...
        qsort( v , sz , sizeof(int) , compare_int );
        ...
}
```

```
void qsort( void *ptr , size_t count , size_t size , int (*cmp)(const void *, const void *) );
```

Passing Functions in

Sorting:

In C, when we want to sort a list of values, we can use the **qsort** function.

sort.c

```
#include <stdio.h>
#include <stdlib.h>
int compare_int( const void *v1 , const void *v2 )
{
	if     ( *(int *)v1<*(int *)v2 ) return -1;
	else if( *(int *)v2<*(int *)v1 ) return  1;
	else                             return  0;
}
int main( void )
{
	…
	qsort( v , sz , sizeof(int) , compare_int );
	…
}
```

void qsort(void *ptr , size_t count , size_t size , int (*cmp)(const void *, const void *));

- **ptr**: a pointer to the first element in the array

Passing Functions in

sort.c

```
#include <stdio.h>
#include <stdlib.h>
int compare_int( const void *v1 , const void *v2 )
{
	if      ( *(int *)v1<*(int *)v2 ) return -1;
	else if( *(int *)v2<*(int *)v1 ) return  1;
	else                              return  0;
}
int main( void )
{
	…
	qsort( v , sz , sizeof(int) , compare_int );
	…
}
```

Sorting:

In C, when we want to sort a list of values, we can use the `qsort` function.

`void qsort( void *ptr , size_t count , size_t size , int (*cmp)(const void *, const void *) );`

- `ptr`: a pointer to the first element in the array
- `count`: the number of elements in the array

Passing Functions in

sort.c

```
#include <stdio.h>
#include <stdlib.h>
int compare_int( const void *v1 , const void *v2 )
{
	if      ( *(int *)v1<*(int *)v2 ) return -1;
	else if( *(int *)v2<*(int *)v1 ) return  1;
	else                              return  0;
}
int main( void )
{
	…
	qsort( v , sz , sizeof(int) , compare_int );
	…
}
```

<u>Sorting</u>:

In C, when we want to sort a list of values, we can use the **qsort** function.

```
void qsort( void *ptr , size_t count , size_t size , int (*cmp)(const void *, const void *) );
```

- **ptr**: a pointer to the first element in the array
- **count**: the number of elements in the array
- **size**: the size of an element

Passing Functions in

sort.c

```
#include <stdio.h>
#include <stdlib.h>
int compare_int( const void *v1 , const void *v2 )
{
	if	( *(int *)v1<*(int *)v2 ) return -1;
	else if( *(int *)v2<*(int *)v1 ) return  1;
	else				 return  0;
}
int main( void )
{
	…
	qsort( v , sz , sizeof(int) , compare_int );
	…
}
```

Sorting:

In C, when we want to sort a list of values, we can use the **qsort** function.

```
void qsort( void *ptr , size_t count , size_t size , int (*cmp)(const void *, const void *) );
```

- **ptr**: a pointer to the first element in the array
- **count**: the number of elements in the array
- **size**: the size of an element
- **cmp**: a pointer to a function taking two **void** pointers and returning an **int**
  - It returns a negative value if the first object pointed to comes before the second.
  - It returns a positive value if the first object pointed to comes after the second.
  - It returns zero if they are “equal”.

Passing Functions in

sort.c

```
#include <stdio.h>
#include <stdlib.h>
int compare_int( const void *v1 , const void *v2 )
{
	if     ( *(int *)v1<*(int *)v2 ) return -1;
	else if( *(int *)v2<*(int *)v1 ) return  1;
	else                             return  0;
}
int main( void )
{
	…
	qsort( v , sz , sizeof(int) , compare_int );
	…
}
```

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

<u>Sorting</u>:

In C, when we want to sort a list of values, we can use the **qsort** function.

void qsort(void *ptr , size_t count , size_t size , int (*cmp

✓Have full control over what gets compared and how the comparison happens

Passing Functions in

sort.c

```c
#include <stdio.h>
#include <stdlib.h>
int compare_int( const void *v1 , const void *v2 )
{
	if      ( *(int *)v1<*(int *)v2 ) return  1;
	else if( *(int *)v2<*(int *)v1 ) return -1;
	else                             return  0;
}
int main( void )
{
	…
	qsort( v , sz , sizeof(int) , compare_int );
	…
}
```

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

<u>Sorting</u>:

In C, when we want to sort a list of values, we can use the **qsort** function.

void qsort(void *ptr , size_t count , size_t size , int (*cmp)(const void *, const void *));

✓Have full control over what gets compared and how the comparison happens

Could change to sorting from largest to smallest by flipping the sign of the return value

Passing Functions in

sort.c

```
#include <stdio.h>
#include <stdlib.h>
int compare_int( const void *v1 , const void *v2 )
{
	if      ( *(int *)v1<*(int *)v2 ) return -1;
	else if( *(int *)v2<*(int *)v1 ) return  1;
	else                              return  0;
}
int main( void )
{
	…
	qsort( v , sz , sizeof(int) , compare_int );
	…
}
```

Sorting:

In C, when we want to sort a list of values, we can use the `qsort` function.

`void qsort( void *ptr , size_t count , size_t size , int (*cmp)(const void *, const void *) );`

- ✓ Have full control over what gets compared and how the comparison happens
- ✗ Lack of overloading requires a single interface that is type-agnostic:
  - ✗ The input array `ptr` has to have type `void *`
  - ✗ Need to provide `size`, the size of an element
  - ✗ The comparison function `cmp` has to take two `void *` arguments
- ✗ The declaration of the type of `cmp` is a mess

Passing Functions in C

<u>Sorting</u>:

In C++, we can make things cleaner/generic by combining overloading and templates:

sort.cpp

```
#include <iostream>
#include <cstdlib>

template< typename T >
int compare_T( const void *v1 , const void *v2 )
{
        if      ( *(const T *)v1<*(const T *)v2 ) return -1;
        else if( *(const T *)v2<*(const T *)v1 ) return  1;
        else                                        return  0;
}

template< typename T >
void my_qsort( T *values , size_t count , int (*cmp)( const void * , const void *) )
{
        qsort( values , count , sizeof(T) , cmp );
}

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_qsort( v , sz , compare_T<int> );

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

        return 0;
}
```

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

Passing Functions in

Sorting:

In C++, we can make things cleaner/generic by combining overloading and templates:

- Use a generic comparator that works for any type supporting "<" comparison

sort.cpp

```
#include <iostream>
#include <cstdlib>

template< typename T >
int compare_T( const void *v1 , const void *v2 )
{
	if     ( *(const T *)v1<*(const T *)v2 ) return -1;
	else if( *(const T *)v2<*(const T *)v1 ) return  1;
	else                                     return  0;
}

template< typename T >
void my_qsort( T *values , size_t count , int (*cmp)( const void * , const void *) )
{
	qsort( values , count , sizeof(T) , cmp );
}

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_qsort( v , sz , compare_T<int> );

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

	return 0;
}
```

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

Passing Functions in C

Sorting:

In C++, we can make things cleaner/generic by combining overloading and templates:

- Use a generic comparator that works for any type supporting "<" comparison
- Define a generic sort interface that uses the template type to determine the size

sort.cpp

```
#include <iostream>
#include <cstdlib>

template< typename T >
int compare_T( const void *v1 , const void *v2 )
{
	if	( *(const T *)v1<*(const T *)v2 ) return -1;
	else if( *(const T *)v2<*(const T *)v1 ) return  1;
	else					 return  0;
}

template< typename T >
void my_qsort( T *values , size_t count , int (*cmp)( const void * , const void *) )
{
	qsort( values , count , sizeof(T) , cmp );
}

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_qsort( v , sz , compare_T<int> );

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

	return 0;
}
```

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

Passing Functions in C

Sorting:

In C++, we can make things cleaner/generic by combining overloading and templates:

- Use a generic comparator that works for any type supporting “<“ comparison
- Define a generic sort interface that uses the template type to determine the size

✓ Invoking the sorting function is cleaner/generic

sort.cpp

```
#include <iostream>
#include <cstdlib>

template< typename T >
int compare_T( const void *v1 , const void *v2 )
{
	if     ( *(const T *)v1<*(const T *)v2 ) return -1;
	else if( *(const T *)v2<*(const T *)v1 ) return  1;
	else                                     return  0;
}

template< typename T >
void my_qsort( T *values , size_t count , int (*cmp)( const void * , const void *) )
{
	qsort( values , count , sizeof(T) , cmp );
}

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_qsort( v , sz , compare_T<int> );

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

	return 0;
}
```

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

Passing Functions in

<u>Sorting</u>:

In C++, we can make things cleaner/generic by combining overloading and templates:

- Use a generic comparator that works for any type supporting “<“ comparison
- Define a generic sort interface that uses the template type to determine the size

✓ Invoking the sorting function is cleaner/generic

✗ Still need to work with **void *** and function pointers

sort.cpp

```
#include <iostream>
#include <cstdlib>

template< typename T >
int compare_T( const void *v1 , const void *v2 )
{
	if     ( *(const T *)v1<*(const T *)v2 ) return -1;
	else if( *(const T *)v2<*(const T *)v1 ) return  1;
	else                                     return  0;
}

template< typename T >
void my_qsort( T *values , size_t count , int (*cmp)( const void * , const void *) )
{
	qsort( values , count , sizeof(T) , cmp );
}

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_qsort( v , sz , compare_T<int> );

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

	return 0;
}
```

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

Passing Functions

<u>Finding the first element</u>:

Consider a simpler case where we want to find the smallest entry in an array.

find_first.cpp

```
#include <iostream>

template< typename T >
bool compare_T( const void *v1 , const void *v2 ){ return *(const T *)v1<*(const T *)v2; }

template< typename T >
unsigned int find_first( T *values , size_t count , bool (*cmp)( const void * , const void *) )
{
	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;

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

	return 0;
}
```

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

Passing Functions

<u>Finding the first element</u>:

Consider a simpler case where we want to find the smallest entry in an array.

- ✓Clean/generic interface

find_first.cpp

```
#include <iostream>

template< typename T >
bool compare_T( const void *v1 , const void *v2 ){ return *(const T *)v1<*(const T *)v2; }

template< typename T >
unsigned int find_first( T *values , size_t count , bool (*cmp)( const void * , const void *) )
{
	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;

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

	return 0;
}
```

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

Passing Functions

<u>Finding the first element</u>:

Consider a simpler case where we want to find the smallest entry in an array.

- ✓ Clean/generic interface
- ✗ Still need to cast and dereference void *
- ✗ Still need to work with function pointers

find_first.cpp

```
#include <iostream>

template< typename T >
bool compare_T( const void *v1 , const void *v2 ){ return *(const T *)v1<*(const T *)v2; }

template< typename T >
unsigned int find_first( T *values , size_t count , bool (*cmp)( const void * , const void *) )
{
	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;

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

	return 0;
}
```

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

Passing Functions

<u>Finding the first element</u>:

The comparator arguments can be templated by the element type.

find_first.cpp

```
#include <iostream>

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

template< typename T >
unsigned int find_first( T *values , size_t count , bool (*cmp)( const T & , const T & ) )
{
	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;

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

	return 0;
}
```

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

Passing Functions

<u>Finding the first element</u>:

The comparator arguments can be templated by the element type.

- ✓ Pass values to directly instead of having to work with void *

find_first.cpp

```
#include <iostream>

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

template< typename T >
unsigned int find_first( T *values , size_t count , bool (*cmp)( const T & , const T & ) )
{
        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;

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

        return 0;
}
```

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

Passing Functions

Finding the first element:

The comparator arguments can be templated by the element type.

- ✓ Pass values to directly instead of having to work with void *
- ✗ Still need to work with function pointers

find_first.cpp

```
#include <iostream>

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

template< typename T >
unsigned int find_first( T *values , size_t count , bool (*cmp)( const T & , const T & ) )
{
	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;

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

	return 0;
}
```

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

Passing Functions

<u>Finding the first element</u>:

We can also template the comparator type (i.e. the function pointer), letting the compiler do the work.

find_first.cpp

```cpp
#include <iostream>

template< typename T >
bool compare_T( const T &t1 , const T &t2 ){ 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;

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

	return 0;
}
```

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

Passing Functions

Finding the first element:

We can also template the comparator type (i.e. the function pointer), letting the compiler do the work.

- ✓Don’t need to work with function pointers
- ✓The implementation is more generic because cmp could be a *functor* – an object acting like a function by defining the operator() operator

find_first.cpp
```
#include <iostream>

template< typename T >
bool compare_T( const T &t1 , const T &t2 ){ 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;

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

	return 0;
}
```

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

Passing Functions

<u>Finding the first element</u>:

We can also template the comparator type (i.e. the function pointer), letting the compiler do the work.

- ✓Don't need to work with function pointers
- ✓The implementation is more generic because `cmp` could be a *functor* – an object acting like a function by defining the `operator()` operator

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( 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;
}
```

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

Outline

- Passing Functions in C vs. C++
- Functors
- Lambdas
- The `auto` keyword

Functors

<u>Definition</u>:

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

Functors

Q: But why do we need functors?

A: To support parametrized functions.

Example:

Suppose we want to find the smallest element, modulo N, where N is a user defined parameter.

- ✗ In C this is hard to do (without global variables or replicating data) because the function only “sees” the arguments, not the value of N.
- ✓ In C++ we can make N be a member data of the functor class.

Functors

Q: But why do we need func

A: To support parametrized

Example:

Suppose we want to find the
user defined parameter.

- ✘ In C this is hard to do (with
the function only "sees" th
- ✓ 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 | ./a.out
 83 86 77 15 93 35 86 92 49 21 62 27
8 -> 49 / 1
>>
```

Outline

- Passing Functions in C vs. C++
- Functors
- Lambdas
- The `auto` keyword

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;
}
```

```
>> ./a.out
 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
be defined outside the sc
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;
}
```

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

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.

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 the 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 the 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 the 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 the brackets
- The arguments to the lambda are described within the parentheses.
- The body/functionality of the lambda is described within the braces.
- The return type of the lambda is derived by the compiler by considering the type returned.

  [WARNING] If there are multiple `return` statements in the body of the functor, they should all return the same type – the compiler won't know which way to cast

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.

<u>Examples</u>:

- 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 o
access is by value or by re

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"

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 | ./a.out
 83 86 77 15 93 35 86 92 49 21 62 27
8 -> 49 / 1
>>
```

Outline

- Passing Functions in C vs. C++
- Functors
- Lambdas
- The `auto` keyword

The **auto** Keyword

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

Q: Given the ability to <u>define</u> a Lambda, how do we <u>declare</u> one?

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

This is unfortunate because we may want to declare the Lambda:

- If we want to use the same Lambda multiple times
- If the Lambda definition takes multiple-lines

The **auto** Keyword

```
[]( 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.

This is unfortunate because we may want to declare the Lambda:

- If we want to use the same Lambda multiple times
- If the Lambda definition takes multiple-lines

The **auto** Keyword

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

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

The **auto** Keyword

<u>auto</u> sort_lambda = [](

When C++ knows an object's ty
declare the object

- When defining an object directly (like a Lambda)

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 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;

	unsigned int 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] << " /
	return 0;
}
```

```
>> echo 6 | ./a.out
 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 **auto** Keyword

auto sort_lambda = [](

When C++ knows an object's ty
declare the object

- When defining an object directly (like a Lambda)

Note:

Like other declarations, we need to have a ";" after the declaration.

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 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;

	unsigned int 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 **auto** Keyword

<u>auto</u> it=c.cbegin()

<u>auto</u> it=v.begin()

When C++ know an object's typ
declare the object

- When defining an object directly (like a Lambda)
- When defining an object indirectly as the return value of a function

print_container.cpp

```
#include <iostream>
#include <vector>

template< class Container >
void print_container( const Container &c )
{
	for( typename Container::const_iterator it=c.cbegin() ; it!=c.cend() ; it ++ )
		std::cout << " " << *it;
	std::cout << std::endl;
}

int main( void )
{
	std::vector< int > v(12);
	for( std::vector< int >::iterator it=v.begin() ; it!=v.end() ; it++ ) *it = rand()%100;
	print_container( v );

	return 0;
}
```

```
>> echo 6 | ./a.out
 83 86 77 15 93 35 86 92 49 21 62 27
>>
```

The **auto** Keyword

<u>auto</u> it=c.cbegin()

<u>auto</u> it=v.begin()

When C++ know an object's typ
declare the object

- When defining an object directly (like a Lambda)
- When defining an object indirectly as the return value of a function

print_container.cpp

```
#include <iostream>
#include <vector>

template< class Container >
void print_container( const Container &c )
{
        for( typename Container::const_iterator it=c.cbegin() ; it!=c.cend() ; it ++ )
                std::cout << " " << *it;
        std::cout << std::endl;
}

int main( void )
{
        std::vector< int > v(12);
        for( std::vector< int >::iterator it=v.begin() ; it!=v.end() ; it++ ) *it = rand()%100;
```

print_container.cpp

```
#include <iostream>
#include <vector>

template< class Container >
void print_container( const Container &c )
{
        for( auto it=c.cbegin() ; it!=c.end() ; it++ )
                std::cout << " " << *it;
        std::cout << std::endl;
}

int main( void )
{
        std::vector< int > v(12);
        for( auto it=v.begin() ; it!=v.end() ; it++ ) *it = rand()%100;
        print_container( v );

        return 0;
}
```

```
>> echo 6 | ./a.out
 83 86 77 15 93 35 86 92 49 21 62 27
>>
```