Functors, Lambdas, and the `std::function` object

Michael Kazhdan
(601.457/657)
Outline

• Recall
• Function passing in C++
• Functors
• Lambdas
• The \texttt{std::function} object
Recall

We often need to support similar functionality for different types.

```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;
}
```

```bash
>> ./compare
1
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.

```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
1
>>
Recall

We often need to support similar functionality for different types.

C++ allows us to handle this better with templates.

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

We often need to support similar functionality for different types.

C++ allows us to handle this better with **templates**.

These are recipes that take are described with 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 better with templates.

These are recipes that take are described with 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.

```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;
}
```

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

• Recall
• Function passing in C++
• Functors
• Lambdas
• The `std::function` object
Function Passing in C++

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

```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.
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).

```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;
}
```

```bash
>> ./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).
   a) This is found by setting the index to the first, iterating through the array, and updating the index if a smaller value is found.
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.

```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 << " \- > " << v[idx] << 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.

```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()
{
    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::endl;

    unsigned int idx = find_first(v, sz, my_comparator< int >);
    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
$ 
```
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.

```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;
}
```

```bash
./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
Functors

Definition:
A functor is an object that acts like a function by defining the function call operator – \texttt{operator()}

Functors

**Definition:**
A functor is an object that acts like a function by defining the call operator – `operator()`.
Functors

Definition:
A functor is an object that acts like a function by defining the function call operator – \texttt{operator()}.

- We wrap the function within an object

```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::cout << std::endl;

    return 0;
}
```

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

Definition:
A functor is an object that acts like a function by defining the call operator – \texttt{operator()}.

- We wrap the function within an object.
- The function is called by treating the object as a function.

```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 << " \rightarrow " << v[idx] << std::endl;

    return 0;
}
```
Functors

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

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

```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: But 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.
- The function should “know” about $N$.
  ✓ In C++ we can make $N$ be a member data of the functor class.
Functors

Q: But 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.
• The function should “know” about $N$.
✓ In C++ we can make $N$ be a member data of the functor class.

```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
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
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.

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

\[
[](\text{int } i_1, \text{int } i_2) \{ \text{return } i_1 < i_2; \}
\]

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

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

• The definition of the lambda is preceded by the brackets

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

```c++
[](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

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

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.
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.
- Most often 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.

**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 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”

```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;
}
```

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

```cpp
#include <iostream>

//find_first_mod.cpp

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.

```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;

    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::endl;
    return 0;
}
```
Outline

• Recall
• Function passing in C++
• Functors
• Lambdas
• The `std::function` object
The \texttt{std::function} Object

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

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

\begin{itemize}
  \item Because the compiler defines it on-the-fly, the type is unspecified.
\end{itemize}

What if we want to \underline{explicitly} declare the lambda:

\begin{itemize}
  \item If we want to use the same lambda multiple times
  \item If the lambda definition takes multiple-lines
\end{itemize}
The std::function Object

```cpp
[](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.

What if we want to explicitly declare the lambda:
- • If we want to use the same lambda multiple times
- • If the lambda definition takes multiple-lines
The `std::function` Object

```cpp
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

```cpp
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

```cpp
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 has to match the type returned by the definition.
The `std::function` Object

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

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

```cpp
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 have to match the argument types in the definition.
The `std::function` object

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.

```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 << " " << 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;
}
```

`find_first_mod.cpp`

```bash
>> 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

```cpp
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::function` Object

```cpp
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.

```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 = [](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

---

```cpp
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;
}
```

```cpp
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;
}
```