C++ Function Pointers, Functors and Lambda Functions

CSCI 1061U — Programming Workshop 2

faisal.qureshi@uoit.ca
Faculty of Science
University of Ontario Institute of Technology
#include <iostream>
using std::cout;
using std::endl;

int square(int x)
{
    return x*x;
}

int main()
{
    int (*fnptr)(int); // function pointer variable
    fnptr = square;
    cout << fnptr(2) << endl;
    return 0;
}

Output 4
Example

```c
void swap(float* a, float* b)
{
    float tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}

void (*m)(float* a, float* b);
m = swap;
```
Example

```c
int neg(int x)
{
    return -1;
}

int (*k)(int x);
k = neg;
```
Example

```c
bool gt(int a, int b)
{
    return a > b;
}

bool (*l)(int a, int b);
l = gt;
```
Example

double sum(double a[], int i, int j, int s) {
    double sum=0.0;
    for int(k=i; k<=j; k+=s) {
        sum += a[k];
    }

    return sum;
}

double (*n)(double a[], int i, int j, int s); n = sum;
Example

class Pt
{
    public:
    int x,y;
};

Pt* create_pt(int x, int y)
{
    Pt* p = new Pt;
    p->x = x;
    p->y = y;
    return p;
}

Pt* (*o)(int x, int y);
o = create_pt;
#include <iostream>
using std::cout;
using std::endl;

int square(int x) {
    return x*x;
}

int neg(int x) {
    return -x;
}

int do_some_process(int x, int (*process)(int)) {
    return process(x);
}

int main() {
    cout << do_some_process(2, square) << endl;
    cout << do_some_process(2, neg) << endl;
    return 0;
}
Function Pointers Uses

- Callback functions
  - Set up listener or callback function that is called when an event occurs (e.g., GUI)

```c
void glutMouseFunc(void (*func)(int button, int state, int x, int y));
```

Enables programmers to provide their own function that will be called whenever there is a mouse event.
Function Pointers

- It is also possible to avoid explicit function pointers by using virtual functions and polymorphism

- Virtual function, however, are implemented behind the scene using function pointers

- Function pointers are often used to pass around processing instructions
C++ Functors

• C++ provide function pointers or *functors*

• Functors are objects that can be used as if these are functions

• Functors are more powerful than good old function pointers, since functors can carry around state

• Functors are only available in C++
#include <iostream>
using std::cout;
using std::endl;

class Square
{
  public:
    int operator()(int x) { return x*x; }
};

int main()
{
  Square a;
  cout << a(3) << endl;

  return 0;
}
Using Functors for Callback

Use template to pass a functor
(remember it is just a class)

class Square
{
    public:
    int operator()(int x) { return x*x; }
};

class Neg
{
    public:
    int operator()(int x) { return -x; }
};

template <typename T>
int do_some_process(int x, T process)
{
    return process(x);
}

int main()
{
    Square sq;
    Neg neg;

cout << do_some_process(2, sq) << endl;
cout << do_some_process(2, neg) << endl;

cout << do_some_process(2, mult_by_5) << endl;

    return 0;
}
Using Functors for Callback

```cpp
class Sum
{
    public:
        int operator()(int x, int y)
        {
            return x + y;
        }
};

template <typename T>
int do_some_process(int x, T process)
{
    return process(x);
}

int main()
{
    Sum sum;
    cout << do_some_process(2, sum) << endl;
    return 0;
}
```

Will this work? No
Function Pointers vs. Functors

```cpp
class Pt {
    public:
        Pt(int x, int y) : _x(x), _y(y) {}

        int _x, _y;
};

void do_some_process(Pt& pt, int v, Pt& (*process)(Pt&, int)) {
    process(pt, v);
}

Example
Pt& add(Pt& pt, int v) {
    pt._x += v;
    return pt;
}

Create another function
Pt& add2(Pt& pt, int v) {
    pt._y += v;
    return pt;
}

But what if we want to add ‘v’ to ‘pt._y’?
```
Function Pointers vs. Functors

class Pt
{
    public:
    Pt(int x, int y) : _x(x), _y(y) {} 
    int _x, _y;
};

template <typename T>
void do_some_process2(Pt& pt, int v, T process)
{
    process(pt, v);
}

class AddFunctor
{
    public:
    AddFunctor(char c) : _c(c) {} 
    Pt& operator()(Pt& pt, int v) {
        if (_c == 'x') pt._x += v;
        else pt._y += v; return pt;
    }
    private:
    char _c;
};

AddFunctor addx('x');
do_some_process2(pt, 10, addy);
AddFunctor addy('y');
do_some_process2(pt, 10, addy);

Uses state stored in the functor to perform them the desired processing.

Unlike function pointers where we needed to create a new function.
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    auto func = [] () { cout << "Hello world." << endl; }
    func();

    return 0;
}

Use the following command to compile

g++ -std=c++11 lambda.cpp
Lambda Functions Syntax

[] ( ) { cout << "Hello world" << endl; }();

[] ( ) { cout << "Hello world" << endl; }();

cout << [] ( ) { return 42; }() << endl;

cout << [] ( ) -> int { return 42; }() << endl;

argument list
function body
invocation

argument list missing
ok if no arguments

return type is missing
ok if compiler can discern it

return type
#include <iostream>
using std::cout;
using std::endl;

int do_some_process(int x, int (*process)(int))
{
    return process(x);
}

int main()
{
    cout << do_some_process(2, [](int x)->int{ return x*x;}) << endl;

    return 0;
}
Variable Capture with Lambda Functions

```cpp
#include <iostream>
#include <string>

using std::cout;
using std::endl;
using std::string;

int main()
{
    string name("Jane");

    [&](){ cout << name << endl; }();

    return 0;
}
```

How did this get here? It was never passed as an argument.

[&] tells the compiler to perform variable capture
Variable Capture with Lambda Function

<table>
<thead>
<tr>
<th>[]</th>
<th>Capture nothing</th>
</tr>
</thead>
<tbody>
<tr>
<td>[&amp;]</td>
<td>Capture any variable reference in the lambda function <em>by reference</em></td>
</tr>
<tr>
<td>[=]</td>
<td>Capture any variable reference in the lambda function <em>by value</em>  (i.e., making a copy)</td>
</tr>
<tr>
<td>[=,&amp;foo]</td>
<td>Capture any variable reference in the lambda function <em>by value</em>  (i.e., making a copy); capture variable foo by reference</td>
</tr>
<tr>
<td>[this]</td>
<td>Capture the ‘this’ pointer of the enclosing class. This means that all members of the enclosing class are available within the lambda function</td>
</tr>
<tr>
<td>[foo]</td>
<td>Capture variable foo by making a copy; do not capture anything else</td>
</tr>
</tbody>
</table>
Lambda Function
Capture By Reference

• Lambda function can modify the values of the captured variable

• Beware of returning the lambda function from a function, since the captured variable might become invalid
Lambda Functions and STL

```cpp
#include <iostream>
#include <vector>
using std::cout;
using std::endl;

int main()
{
    std::vector<int> v;

    for (int i=0; i<10; ++i) v.push_back(i*2);

    std::for_each(v.begin(), v.end(), [](int val){ cout << val << endl; });

    return 0;
}
```
Exercise

Implementing a general purpose find_smallest() method
bool smaller(int i, int j)
{
    return i < j;
}

int find_smallest(int a[], int n)
{
    int smallest = a[0];
    for (int i=1; i<n; ++i) {
        if (smaller(a[i], smallest)) smallest = a[i];
    }
    return smallest;
}

int main()
{
    srand(0);
    int a[6];
    for (int i=0; i<6; ++i) a[i] = rand();

    for (int i=0; i<6; ++i) cout << a[i] << endl;

    int smallest = find_smallest(a, 6);
    cout << "smallest = " << smallest << endl;

    return 0;
}
Exercise

Implementing a general purpose find_smallest() method

Available on the course web
Due in class
Submit via Blackboard