28. Pointers and Functions (DIY)

Objectives

- Understand function pointer values
- Understand function pointers
- Understand how function pointers allow us to write compact and flexible code

Functions and Pointers

Try this:

```
#include <iostream>

double avg(int x, int y)
{
    return (x + y) / 2.0;
}

int main()
{
    std::cout << avg(1, 2) << std::endl;

    return 0;
}
```

No surprises. Of course

```
avg(1, 2)
```

is a function call. No big deal. BUT … now try this:

```
#include <iostream>

double avg(int x, int y)
{
    return (x + y) / 2.0;
}

int main()
{
    std::cout << avg(1, 2) << std::endl;
    std::cout << (unsigned int) avg << std::endl;

    return 0;
}
```

What do you get? What do you think `avg` means?

`avg` is actually the address of the function `avg`. You can think of it this way: Not only data (i.e. variable values) lives in your computer's memory, even code lives in your RAM. `avg` is the address of the first line of code of `avg`.

You can save the function address in a variable. Such a variable is called **function pointer**. Here's an example:

```
#include <iostream>

double avg(int x, int y)
{
    return (x + y) / 2.0;
```

```
}

int main()
{
    std::cout << avg(1, 2) << std::endl;
    std::cout << (unsigned int) avg << std::endl;

    double (*g)(int, int);
    g = &avg;
    std::cout << (unsigned int) g << std::endl;

    return 0;
}
```

Note the declaration of `g`.

Exercise. Instead of this:

```
...
    double (*g)(int, int);
...
```

can you do this:

```
...
    double *g(int, int);
...
```

Why?

Exercise. It turns out that instead of

```
...
    g = &avg;
...
```

you can also write

```
...
    g = avg;
...
```

In other words, for the case of functions, the address operator & is optional.

What can you do with a function pointer?

So what can you do with a function pointer? Well … you can use it just like a function:

```
#include <iostream>

double avg(int x, int y)
{
    return (x + y) / 2.0;
}

int main()
{
    std::cout << avg(1, 2) << std::endl;
    std::cout << (unsigned int) avg << std::endl;

    double (*g)(int, int);
    g = &avg;
    std::cout << g(1, 2) << std::endl;
    std::cout << (unsigned int) g << std::endl;

    return 0;
}
```

Exercise. Write a function `square()` that accepts a value (a `double`) and returns the square of the value. Create a function pointer `p` and store the address of `square` in `p`. Using `p`, print the square of `2`.

Why?

OK. The above looks bizarre. But what good is it? Why not just use `avg`?

Because now even functions can be viewed as values. The following will make things clear.

For instance suppose you have two functions `f1`, `f2`, both accepting two integers and returning a `double` and you have a program that looks like where `f1`, `f2` are used a lot but in two different scenarios:

```
if (x < 0)
{
    double a = f1(x + a, b);
    double b = f1(a, b + c / 2);
    double c = a + b + f1(5.0, a + b);
    z = a + b + c + f1(1, 1);
}
else
{
    double a = f2(x + a, b);
    double b = f2(a, b + c / 2);
    double c = a + b + f2(5.0, a + b);
    z = a + b + c + f2(1, 1);
}
std::cout << z << '\n';
```

Note that the computation in both case are essentially the same other than a difference in which of `f1`, `f2` should be used. Now if your programming language has the concept of function pointers like C++, you can do this:

```
double (*g)(int, int) = (x < 0 ? &f1 : &f2);
double a = g(x + a, b);
double b = g(a, b + c / 2);
double c = a + b + g(5.0, a + b);
z = a + b + c + g(1, 1);

std::cout << z << '\n';
```

Note in particular, you can create a function that accepts functions as parameters! If you think of functions as computing with inputs values such as integer and doubles, then you can also think of functions as performing computations on functions … functions operating on functions!!!

Are there such things in real life? Of course. Suppose you want to find the maximum value of a function like the function x*x + x - 5 in the interval [a, b]. You might do this:

```
double max(double a, double b)
{
    double m = a * a + a - 5;
    for (double x = a + 0.001; x <= b; x += 0.001)
    {
```

```
        double y = x * x + x - 5;
        if (y  > m)
        {
            m = y;
        }
    }
    return m;
}
```

Unfortunately, the function x * x + x – 5 is "hard-coded" into the above `max()` function. It cannot be used to compute for instance the maximum value of 3 * x * x * x – 5 * x * x + 12. You would have to write another function, say `max2()`, that does more or less the same thing.

But … ahhh … since C/C++ allows function pointers, you can do this:

```
double max(double a, double b, double (*f) (double))
{
    double m = f(a);
    for (double x = a + 0.001; x < b; x += 0.001)
    {
        double y = f(x);
        if (y > m)
        {
            m = y;
        }
    }
    double y = f(b);
    if (y > m)
    {
        m = y;
    }
    return m;
}
```

Now if you want to compute the maximum of 3 * x * x * x – 5 * x * x + 12 for 1<= x <= 100, all you need to do is to do this:

```
double f(double x)
{
    return 3 * x * x * x - 5 * x * x + 12;
}
```

and then call the above `max` function with this `f`:

```
std::cout << max(1, 100, &f) << std::endl;
```

Exercise. The above `max` function tests x points spaced out in such a way that consecutive x values differ by 0.001. Write a max function that lets you specify this gap, called `dx`, where the default value is `0.001`.

Exercise. Write a `min` function that does the obvious thing similar to the above.

Exercise. Write an area function that computes the area of a function `f`.

For instance `area(1, 5, &f, 0.0001)` computes the area under the curve y = f(x) from x = 1 to x = 5 using the sum of tiny rectangular strips of width 0.0001. Let the default width be 0.001.

Exercise. Look at your bubblesort function again:

```
void bubblesort(int x[], int size);
```

The problem is that it sorts either in ascending or descending order. Note that The difference is only in comparison of values in array x. You need to know if x[i] > x[i+1] is true. We can have two boolean comparison functions:

```
bool lessthan(int x, int y);
bool greaterthan(int x, int y);
```

and write a `bubblesort` function so that you can call

```
bubblesort(x, size, &lessthan);
```

to sort `x` in ascending order and call

```
bubblesort(x, size, &greaterthan);
```

to sort `x` in descending order.

Array of function pointers

2022/10/06: While working on the “robot assignment” of CISS245, Eric Garcia asked about using an array of function pointers to simplify the code. This section was then added.

In the above I let g be one of two functions:

```
double (*g)(int, int) = (x < 0 ? &f1 : &f2);
...
```

What if you have a collection of 3 functions? Or 5 functions? Or 100 functions? You can put them into an array of function pointers, and then select which one you want by specifying an index value. Run this:

```
#include <iostream>

int f(int x, int y)
{
    return x;
}

int g(int x, int y)
{
    return y;
}

int h(int x, int y)
{
    return x + y;
}

int main()
{
    int (*F[3])(int, int) = {f, g, h};
    for (int i = 0; i < 3; ++i)
    {
        std::cout << F[i](0, 1) << '\n';
    }

    return 0;
}
```