# 7.8 — Function Pointers

BY ALEX ON AUGUST 8TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

In lesson **6.7 -- Introduction to pointers**, you learned that a pointer is a variable that holds the address of another variable. Function pointers are similar, except that instead of pointing to variables, they point to functions!

Consider the following function:

```
int foo()
{
    return 5;
}
```

Identifier foo is the function's name. But what type is the function? Functions have their own l-value function type -- in this case, a function type that returns an integer and takes no parameters. Much like variables, functions live at an assigned address in memory.

When a function is called (via the () operator), execution jumps to the address of the function being called:

```
int foo() // code for foo starts at memory address 0x002717f0
{
    return 5;
}

int main()
{
    foo(); // jump to address 0x002717f0

    return 0;
}
```

At some point in your programming career (if you haven't already), you'll probably make a simple mistake:

```
#include <iostream>

int foo() // code starts at memory address 0x002717f0
{
    return 5;
}

int main()
{
    std::cout << foo << '\n'; // we meant to call foo(), but instead we're printing foo itself

    return 0;
}
```

Instead of calling function foo() and printing the return value, we've unintentionally sent function foo directly to std::cout. What happens in this case?

On the author's machine, this printed:

```
0x002717f0
```

...but it may print some other value (e.g. 1) on your machine, depending on how your compiler decides to convert the function pointer to another type for printing. If your machine doesn't print the function's address, you may be able to force it to do so by converting the function to a void pointer and printing that:

```
#include <iostream>

int foo() // code starts at memory address 0x002717f0
{
    return 5;
}

int main()
{
    std::cout << reinterpret_cast<void*>(foo) << '\n'; // Tell C++ to interpret function foo a

    return 0;
}
```

Just like it is possible to declare a non-constant pointer to a normal variable, it's also possible to declare a non-constant pointer to a function. In the rest of this lesson, we'll examine these function pointers and their uses. Function pointers are a fairly advanced topic, and the rest of this lesson can be safely skipped or skimmed by those only looking for C++ basics.

**Pointers to functions**

The syntax for creating a non-const function pointer is one of the ugliest things you will ever see in C++:

```
// fcnPtr is a pointer to a function that takes no arguments and returns an integer
int (*fcnPtr)();
```

In the above snippet, fcnPtr is a pointer to a function that has no parameters and returns an integer. fcnPtr can point to any function that matches this type.

The parenthesis around *fcnPtr are necessary for precedence reasons, as `int *fcnPtr()` would be interpreted as a forward declaration for a function named fcnPtr that takes no parameters and returns a pointer to an integer.

To make a const function pointer, the const goes after the asterisk:

```
int (*const fcnPtr)();
```

If you put the const before the int, then that would indicate the function being pointed to would return a const int.

**Assigning a function to a function pointer**

Function pointers can be initialized with a function (and non-const function pointers can be assigned a function):

```
int foo()
{
    return 5;
}

int goo()
{
    return 6;
}

int main()
{
    int (*fcnPtr)(){ foo }; // fcnPtr points to function foo
    fcnPtr = goo; // fcnPtr now points to function goo

    return 0;
}
```

One common mistake is to do this:

```
fcnPtr = goo();
```

This would actually assign the return value from a call to function goo() to fcnPtr, which isn't what we want. We want fcnPtr to be assigned the address of function goo, not the return value from function goo(). So no parenthesis are needed.

Note that the type (parameters and return type) of the function pointer must match the type of the function. Here are some examples of this:

```
// function prototypes
int foo();
double goo();
int hoo(int x);

// function pointer assignments
int (*fcnPtr1)(){ foo }; // okay
int (*fcnPtr2)(){ goo }; // wrong -- return types don't match!
double (*fcnPtr4)(){ goo }; // okay
fcnPtr1 = hoo; // wrong -- fcnPtr1 has no parameters, but hoo() does
int (*fcnPtr3)(int){ hoo }; // okay
```

Unlike fundamental types, C++ *will* implicitly convert a function into a function pointer if needed (so you don't need to use the address-of operator (&) to get the function's address). However, it will not implicitly convert function pointers to void pointers, or vice-versa.

**Calling a function using a function pointer**

The other primary thing you can do with a function pointer is use it to actually call the function. There are two ways to do this. The first is via explicit dereference:

```
int foo(int x)
{
    return x;
}

int main()
{
    int (*fcnPtr)(int){ foo }; // assign fcnPtr to function foo
    (*fcnPtr)(5); // call function foo(5) through fcnPtr.

    return 0;
}
```

The second way is via implicit dereference:

```
int foo(int x)
{
    return x;
}

int main()
{
    int (*fcnPtr)(int){ foo }; // assign fcnPtr to function foo
    fcnPtr(5); // call function foo(5) through fcnPtr.

    return 0;
}
```

As you can see, the implicit dereference method looks just like a normal function call -- which is what you'd expect, since normal function names are pointers to functions anyway! However, some older compilers do not support the implicit dereference method, but all modern compilers should.

One interesting note: Default parameters won't work for functions called through function pointers. Default parameters are resolved at compile-time (that is, if you don't supply an argument for a defaulted parameter, the compiler substitutes one in for you when the code is compiled). However, function pointers are resolved at run-time. Consequently, default parameters can not be resolved when making a function call with a function pointer. You'll explicitly have to pass in values for any defaulted parameters in this case.

**Passing functions as arguments to other functions**

One of the most useful things to do with function pointers is pass a function as an argument to another function. Functions used as arguments to another function are sometimes called **callback functions**.

Consider a case where you are writing a function to perform a task (such as sorting an array), but you want the user to be able to define how a particular part of that task will be performed (such as whether the array is sorted in ascending or descending order). Let's take a closer look at this problem as applied specifically to sorting, as an example that can be generalized to other similar problems.

Many comparison-based sorting algorithms work on a similar concept: the sorting algorithm iterates through a list of numbers, does comparisons on pairs of numbers, and reorders the numbers based on the results of those comparisons. Consequently, by varying the comparison, we can change the way the algorithm sorts without affecting the rest of the sorting code.

Here is our selection sort routine from a previous lesson:

```
#include <algorithm> // for std::swap, use <utility> instead if C++11

void SelectionSort(int *array, int size)
{
    // Step through each element of the array
    for (int startIndex{ 0 }; startIndex < (size - 1); ++startIndex)
    {
        // smallestIndex is the index of the smallest element we've encountered so far.
        int smallestIndex{ startIndex };

        // Look for smallest element remaining in the array (starting at startIndex+1)
        for (int currentIndex{ startIndex + 1 }; currentIndex < size; ++currentIndex)
        {
            // If the current element is smaller than our previously found smallest
            if (array[smallestIndex] > array[currentIndex]) // COMPARISON DONE HERE
                // This is the new smallest number for this iteration
                smallestIndex = currentIndex;
        }

        // Swap our start element with our smallest element
        std::swap(array[startIndex], array[smallestIndex]);
    }
}
```

Let's replace that comparison with a function to do the comparison. Because our comparison function is going to compare two integers and return a boolean value to indicate whether the elements should be swapped, it will look something like this:

```
bool ascending(int x, int y)
{
    return x > y; // swap if the first element is greater than the second
}
```

And here's our selection sort routine using the ascending() function to do the comparison:

```
#include <algorithm> // for std::swap, use <utility> instead if C++11

void SelectionSort(int *array, int size)
```

```
{
    // Step through each element of the array
    for (int startIndex{ 0 }; startIndex < (size - 1); ++startIndex)
    {
        // smallestIndex is the index of the smallest element we've encountered so far.
        int smallestIndex{ startIndex };

        // Look for smallest element remaining in the array (starting at startIndex+1)
        for (int currentIndex{ startIndex + 1 }; currentIndex < size; ++currentIndex)
        {
            // If the current element is smaller than our previously found smallest
            if (ascending(array[smallestIndex], array[currentIndex])) // COMPARISON DONE HERE
                // This is the new smallest number for this iteration
                smallestIndex = currentIndex;
        }

        // Swap our start element with our smallest element
        std::swap(array[startIndex], array[smallestIndex]);
    }
}
```

Now, in order to let the caller decide how the sorting will be done, instead of using our own hard-coded comparison function, we'll allow the caller to provide their own sorting function! This is done via a function pointer.

Because the caller's comparison function is going to compare two integers and return a boolean value, a pointer to such a function would look something like this:

```
bool (*comparisonFcn)(int, int);
```

So, we'll allow the caller to pass our sort routine a pointer to their desired comparison function as the third parameter, and then we'll use the caller's function to do the comparison.

Here's a full example of a selection sort that uses a function pointer parameter to do a user-defined comparison, along with an example of how to call it:

```
#include <algorithm> // for std::swap, use <utility> instead if C++11
#include <iostream>

// Note our user-defined comparison is the third parameter
void selectionSort(int *array, int size, bool (*comparisonFcn)(int, int))
{
    // Step through each element of the array
    for (int startIndex{ 0 }; startIndex < (size - 1); ++startIndex)
    {
        // bestIndex is the index of the smallest/largest element we've encountered so far.
        int bestIndex{ startIndex };

        // Look for smallest/largest element remaining in the array (starting at startIndex+1)
        for (int currentIndex{ startIndex + 1 }; currentIndex < size; ++currentIndex)
        {
            // If the current element is smaller/larger than our previously found smallest
            if (comparisonFcn(array[bestIndex], array[currentIndex])) // COMPARISON DONE HERE
                // This is the new smallest/largest number for this iteration
                bestIndex = currentIndex;
        }

        // Swap our start element with our smallest/largest element
        std::swap(array[startIndex], array[bestIndex]);
    }
}

// Here is a comparison function that sorts in ascending order
```

```
// (Note: it's exactly the same as the previous ascending() function)
bool ascending(int x, int y)
{
    return x > y; // swap if the first element is greater than the second
}

// Here is a comparison function that sorts in descending order
bool descending(int x, int y)
{
    return x < y; // swap if the second element is greater than the first
}

// This function prints out the values in the array
void printArray(int *array, int size)
{
    for (int index{ 0 }; index < size; ++index)
        std::cout << array[index] << ' ';
    std::cout << '\n';
}

int main()
{
    int array[9]{ 3, 7, 9, 5, 6, 1, 8, 2, 4 };

    // Sort the array in descending order using the descending() function
    selectionSort(array, 9, descending);
    printArray(array, 9);

    // Sort the array in ascending order using the ascending() function
    selectionSort(array, 9, ascending);
    printArray(array, 9);

    return 0;
}
```

This program produces the result:

```
9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9
```

Is that cool or what? We've given the caller the ability to control how our selection sort does its job.

The caller can even define their own "strange" comparison functions:

```
bool evensFirst(int x, int y)
{
    // if x is even and y is odd, x goes first (no swap needed)
    if ((x % 2 == 0) && !(y % 2 == 0))
        return false;

    // if x is odd and y is even, y goes first (swap needed)
    if (!(x % 2 == 0) && (y % 2 == 0))
        return true;

        // otherwise sort in ascending order
    return ascending(x, y);
}

int main()
{
    int array[9]{ 3, 7, 9, 5, 6, 1, 8, 2, 4 };
```

```
18
19      selectionSort(array, 9, evensFirst);
20      printArray(array, 9);
21
22      return 0;
23  }
```

The above snippet produces the following result:

```
2 4 6 8 1 3 5 7 9
```

As you can see, using a function pointer in this context provides a nice way to allow a caller to "hook" their own functionality into something you've previously written and tested, which helps facilitate code reuse! Previously, if you wanted to sort one array in descending order and another in ascending order, you'd need multiple versions of the sort routine. Now you can have one version that can sort any way the caller desires!

Note: If a function parameter is of a function type, it will be converted to a pointer to the function type. This means

```
1  void selectionSort(int *array, int size, bool (*comparisonFcn)(int, int))
```

can be equivalently written as:

```
1  void selectionSort(int *array, int size, bool comparisonFcn(int, int))
```

This only works for function parameters, not stand-alone function pointers, and so is of somewhat limited use.

**Providing default functions**

If you're going to allow the caller to pass in a function as a parameter, it can often be useful to provide some standard functions for the caller to use for their convenience. For example, in the selection sort example above, providing the ascending() and descending() function along with the selectionSort() function would make the caller's life easier, as they wouldn't have to rewrite ascending() or descending() every time they want to use them.

You can even set one of these as a default parameter:

```
1  // Default the sort to ascending sort
2  void selectionSort(int *array, int size, bool (*comparisonFcn)(int, int) = ascending);
```

In this case, as long as the user calls selectionSort normally (not through a function pointer), the comparisonFcn parameter will default to ascending.

**Making function pointers prettier with typedef or type aliases**

Let's face it -- the syntax for pointers to functions is ugly. However, typedefs can be used to make pointers to functions look more like regular variables:

```
1  typedef bool (*validateFcn)(int, int);
```

This defines a typedef called "validateFcn" that is a pointer to a function that takes two ints and returns a bool.

Now instead of doing this:

```
1  bool validate(int x, int y, bool (*fcnPtr)(int, int)); // ugly
```

You can do this:

```
1  bool validate(int x, int y, validateFcn pfcn) // clean
```

Which reads a lot nicer! However, the syntax to define the typedef itself can be difficult to remember.

In C++11, you can instead use type aliases to create aliases for function pointers types:

```
using validateFcn = bool(*)(int, int); // type alias
```

This reads more naturally than the equivalent typedef, since the name of the alias and the alias definition are placed on opposite sides of the equals sign.

Using a type alias is identical to using a typedef:

```
bool validate(int x, int y, validateFcn pfcn) // clean
```

**Using std::function in C++11**

Introduced in C++11, an alternate method of defining and storing function pointers is to use std::function, which is part of the standard library <functional> header. To define a function pointer using this method, declare a std::function object like so:

```
#include <functional>
bool validate(int x, int y, std::function<bool(int, int)> fcn); // std::function method that re
```

As you see, both the return type and parameters go inside angled brackets, with the parameters inside parenthesis. If there are no parameters, the parentheses can be left empty. Although this reads a little more verbosely, it's also more explicit, as it makes it clear what the return type and parameters expected are (whereas the typedef method obscures them).

Updating our earlier example with std::function:

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

int foo()
{
    return 5;
}

int goo()
{
    return 6;
}

int main()
{
    std::function<int()> fcnPtr{ foo }; // declare function pointer that returns an int and ta
    fcnPtr = goo; // fcnPtr now points to function goo
    std::cout << fcnPtr() << '\n'; // call the function just like normal

    return 0;
}
```

Note that you can also type alias std::function:

```
using validateFcnRaw = bool(*)(int, int); // type alias to raw function pointer
using validateFcn = std::function<bool(int, int)>; // type alias to std::function
```

**Type inference for function pointers**

Much like the *auto* keyword can be used to infer the type of normal variables, the *auto* keyword can also infer the type of a function pointer.

```
#include <iostream>

int foo(int x)
{
    return x;
```

```
}

int main()
{
    auto fcnPtr{ foo };
    std::cout << fcnPtr(5) << '\n';

    return 0;
}
```

This works exactly like you'd expect, and the syntax is very clean. The downside is, of course, that all of the details about the function's parameters types and return type are hidden, so it's easier to make a mistake when making a call with the function, or using its return value.

**Conclusion**

Function pointers are useful primarily when you want to store functions in an array (or other structure), or when you need to pass a function to another function. Because the native syntax to declare function pointers is ugly and error prone, we recommend using std::function. In places where a function pointer type is only used once (e.g. a single parameter or return value), std::function can be used directly. In places where a function pointer type is used multiple times, a type alias to a std::function is a better choice (to prevent repeating yourself).

**Quiz time!**

1) In this quiz, we're going to write a version of our basic calculator using function pointers.

1a) Create a short program asking the user for two integer inputs and a mathematical operation ('+', '-', '*', '/'). Ensure the user enters a valid operation.

**Show Solution**

1b) Write functions named add(), subtract(), multiply(), and divide(). These should take two integer parameters and return an integer.

**Show Solution**

1c) Create a type alias named arithmeticFcn for a pointer to a function that takes two integer parameters and returns an integer. Use std::function.

**Show Solution**

1d) Write a function named getArithmeticFunction() that takes an operator character and returns the appropriate function as a function pointer.

**Show Solution**

1e) Modify your main() function to call getArithmeticFunction(). Call the return value from that function with your inputs and print the result.

**Show Solution**

Here's the full program:

**Show Solution**

**7.9 -- The stack and the heap**

**Index**

**7.7 -- Default arguments**

C++ TUTORIAL | PRINT THIS POST

## 439 comments to 7.8 — Function Pointers

« Older Comments 1 ... 4 5 6

Fabio Henrique
February 3, 2020 at 7:45 am · Reply

Hi there !
I have a question, in line 66 of the full program:

```
1 arithmeticFcn fcn{ getArithmeticFcn(op) };
```

I successfully wrote the entire program but I used `auto` for this variable type on this line. I'm wrong to do this ?!
I did because I thought it was pretty clear that this variable would get the return type of `getArithmeticFunction()`
And I named the variable as `fcn_ptr` to give even more clues of it's return type.
Let me know if I'm wrong, please.
Thank you !

nascardriver
February 4, 2020 at 8:30 am · Reply

That's fine.

Mn3m
February 1, 2020 at 2:09 am · Reply

A very simple demonstration of function pointers used to allow the user to choose the sorting type, but with bubble sort.

```
#include <iostream>
typedef bool(*FunctionPointer)(int a, int b);
typedef void(*SortPointer)(int[], int, FunctionPointer);// Variable names aren't necessary









void sort(int array[], int length, FunctionPointer functionPtr)
{


    for (int i = 0; i < length - 1; ++i)
    {


        for (int j = 0; j < length-1; ++j)
        {
            if (functionPtr(array[j], array[j + 1]))
            std::swap(array[j], array[j + 1]);
        }
    }

}

bool ascending(int a, int b)
{
    return a > b;
}



bool descending(int a, int b)
{
    return a < b;
}




int main()
{
    int choice = 0;
    constexpr int length = 9;
    int array[]{ 6, 79, 2, 45, 99, 1, 102, 555, 1337};
    SortPointer sortPtr{ sort };//Initialized.
    FunctionPointer functionPtr;//To be assigned.

    std::cout << "[1]Ascending\n[2]Descending\n";
    std::cout << "Choice: ";
    std::cin >> choice;
    if (choice == 1)
    {
        functionPtr = ascending;
    }

    else
```

```
    {
        functionPtr = descending;
    }
    sortPtr(array, length, functionPtr);


    for (int i = 0; i < length; ++i)
    {
        std::cout << array[i] << ' ';
    }



    return 0;
}
```

nascardriver
February 1, 2020 at 3:09 am · Reply

Good observation! `std::sort` also allows you to do exactly this.

```
std::sort(std::begin(array), std::end(array), functionPtr);
```

Mn3m
February 1, 2020 at 3:37 am · Reply

Really appreciate your feedback for both comments, thank you, pal.

josecyc
January 31, 2020 at 3:27 pm · Reply

The func pointer definition you provide doesn't seem to be correct to compile, is there something that I'm missing?

I'm getting the following error:

```
error: expected ';' at end of declaration
        int (*fcnPtr)()
                      ^
                      ;
```

Here's my code:

```
#include <iostream>

int foo()
{
    return 5;
}

int goo()
{
    return 6;
}

int main()
{

```

```
    int (*fcnPtr)(){ foo }; // initializing fcnPtr pointer to point to function foo
    fcnPtr = goo;           // assigning fcnPtr to point to function goo

    return 0;
}
```

[SOLVED]
Edit: I found that initializing a function pointer in line is only possible in c++11, so compiling with -std=c++11 is needed, this solved it although it still left me wondering why wasn't this implemented in the original c++

nascardriver
February 1, 2020 at 3:05 am · Reply

Brace initialization was added in C++11. Before that, you had to use copy or direct initialization.

```
int (*fcnPtr)() = foo;
int (*fcnPtr)()(foo);
```

Enable and use the highest standard available (C++17), you'll run into more issues if you use old standards.

Mn3m
January 31, 2020 at 9:24 am · Reply

```
#include <iostream>
#include<functional>
using ArithmeticFcn = std::function <int(int, int)>;









int getInput()
{
    int value;
    std::cin >> value;
    return value;
}


char getOperation()
{
    char Operator;
    do
    {

     std::cin.ignore(32767, '\n');// The line below will execute more than once if the use
     std::cout << "Choose an operation, +, -, *, /";
     std::cin >> Operator;

    } while (Operator != '+' && Operator != '-' && Operator != '*' && Operator != '/');
```

```
34        return Operator;
35    }
36
37
38    int add(int a, int b)
39    {
40
41        return a + b;
42    }
43
44
45
46    int subtract(int a, int b)
47    {
48
49        return a - b;
50    }
51
52
53
54    int multiply(int a, int b)
55    {
56
57        return a * b;
58    }
59
60
61
62    int divide(int a, int b)
63    {
64
65        return a / b;
66    }
67
68
69    ArithmeticFcn getArithmeticFunction(char Operator)
70    {
71        ArithmeticFcn fcn;
72        switch (Operator)
73        {
74
75        case '+':
76        return  fcn = add ;
77
78        case '-':
79        return fcn = subtract ;
80
81
82        case '*':
83        return fcn = multiply ;
84
85
86        case '/':
87        return fcn = divide ;
88
89        default:
90        return nullptr; // Already ensured not to get an undesired value by getOperation(), bu
91
92        }
93
94    }
95
```

```
96
97
98
99  int main()
100 {
101
102     std::cout << "Enter an integer: ";
103     int value1 = getInput();
104     std::cout << "Enter another integer: ";
105     int value2 = getInput();
106     char Operator = getOperation();
107     ArithmeticFcn fcn{ getArithmeticFunction(Operator) };
108     std::cout<<fcn(value1, value2);
109     return 0;
110 }
```

nascardriver
February 1, 2020 at 3:02 am · Reply

You don't need to create a variable if you don't use it. Line 76-87 can return the functions right away.
Always assume that a function gets called with any value that's valid for its parameter types. Although you only call `getArithmeticFunction` with the result of `getOperation`, that could change in the future or you could have a bug in `getOperation`. Adding a default case was the right choice.
Variables should be initialized with brace initialization for higher type safety. See lesson 1.4.

cnoob
January 4, 2020 at 6:29 am · Reply

Another question:
Why does this call:

```
1 std::cout << getArithmeticFcn(getOperator())(getInteger(), getInteger());
```

reverse the order of the arithmetic operations? (I mean it executes y-x, instead of x-y for example.)

(p.s.:I do not get any notification emails from this forum.)

nascardriver
January 5, 2020 at 6:34 am · Reply

The order of evaluation of arguments is unspecified. If you need a specific order, call `getInteger` earlier and store the results in variables.

Check your spam inbox. Your email address is set, you should receive notifications. If there are no notifications in spam, please point it out again.

cnoob
January 5, 2020 at 10:13 am · Reply

Its very good to know, thank you! I still couldnt find email notifications though, neither in the spam inbox nor anywhere else.

Alex

January 8, 2020 at 9:47 am · Reply

I checked the mail log and here's what I see (I've sanitized the IP/hostname):

status=deferred (host fmx.somewhere.hu[1.2.3.4] said: 450 4.7.1 Spam suspect e-mail #406 [bfb743e5]

It looks like your host is blocking emails from this site as suspected spam, probably because they use a templated format.

Ayushman Singh
December 17, 2019 at 8:45 am · Reply

Is this legal:

```
1
2
    std::cout<< "\n: "<<getArithmeticFunction(operate)(input1, input2);
```

than of

```
1
2
3
arithmeticFcn fcn{ getArithmeticFcn(op) };
   std::cout << x << ' ' << op << ' ' << y << " = " << fcn(x, y) << '\n';
```

nascardriver
December 17, 2019 at 8:47 am · Reply

Yes, same thing, but harder to read.

Ged
December 8, 2019 at 3:23 pm · Reply

Both codes work.
1. Wouldn't it be better to use std::function for reading purposes?
2. Didn't you say to avoid global variables or because we are only using one file this is a viable option?

3. When this code gets executed it starts the function from the right and division gets flipped. For example it gets y value first and only then x. So y becomes x and x becomes y. Earlier you said the program can start the function from the left or the right randomly. Is there any way to tell the program to start the function from the left or do I need to initialise 2 variables and then put them inside the function?

```
1
2
3
4
5
//
//
std::cout << arithmeticFcn(getInteger(), getInteger()) << '\n';
//
//
```

```
1
2
3
4
5
6
7
8
9
10
11
12
// quiz number one

#include <iostream>
#include <functional>

char getOperator()
{
    char sym{};
    while (true)
    {
        std::cout << "Enter +, -, *, / ";
        std::cin >> sym;
```

```
        std::cin.ignore(32767, '\n');
        if (sym == '+' || sym == '-' || sym == '*' || sym == '/')
            return sym;
        else
            std::cout << "Wrong input, please try again" << '\n';
    }
}

int getInteger()
{
    int integer{};
    while(true)
    {
        std::cout << "Enter an integer ";
        std::cin >> integer;

        if (std::cin.fail())
        {
            std::cin.clear();
            std::cin.ignore(32767, '\n');
            std::cout << "Wrong input, please try again" << '\n';
        }
        else
        {
            std::cin.ignore(32767, '\n');
            return integer;
        }
    }
}

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

int subtract(int x, int y)
{
    return x - y;
}

int multiply(int x, int y)
{
    return x * y;
}

int divide(int x, int y)
{
    return x / y;
}

std::function<int(int, int)> getArithmeticFunction(char c)
{
    switch (c)
    {
    case '+':    return add;
    case '-':    return subtract;
    case '*':    return multiply;
    case '/':    return divide;
    }
}

int main()
```

```
{
    std::function<int(int, int)> arithmeticFcn{ getArithmeticFunction(getOperator()) };
    std::cout << arithmeticFcn(getInteger(), getInteger()) << '\n';

    return 0;
}
```

```
// quiz number two

#include <iostream>
#include <functional>
#include <iterator>

char getOperator()
{
    char sym{};
    while (true)
    {
        std::cout << "Enter +, -, *, / ";
        std::cin >> sym;
        std::cin.ignore(32767, '\n');
        if (sym == '+' || sym == '-' || sym == '*' || sym == '/')
            return sym;
        else
            std::cout << "Wrong input, please try again" << '\n';
    }
}

int getInteger()
{
    int integer{};
    while (true)
    {
        std::cout << "Enter an integer ";
        std::cin >> integer;

        if (std::cin.fail())
        {
            std::cin.clear();
            std::cin.ignore(32767, '\n');
            std::cout << "Wrong input, please try again" << '\n';
        }
        else
        {
            std::cin.ignore(32767, '\n');
            return integer;
        }
    }
}

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

int subtract(int x, int y)
{
    return x - y;
}

int multiply(int x, int y)
{
```

```
56      return x * y;
57  }
58
59  int divide(int x, int y)
60  {
61      return x / y;
62  }
63
64  struct arithmeticStruct
65  {
66      char mathematicalOperator;
67      std::function<int(int, int)> arithmeticFunction;
68  };
69
70  static const arithmeticStruct arithmeticArray[]
71  {
72      {'+', add},
73      {'-', subtract},
74      {'*', multiply},
75      {'/', divide},
76  };
77
78  std::function<int(int, int)> getArithmeticFunction(char c)
79  {
80      for (const auto& symbol : arithmeticArray)
81      {
82          if (c == symbol.mathematicalOperator)
83          {
84              return symbol.arithmeticFunction;
85          }
86      }
87  }
88
89  int main()
90  {
91      std::function<int(int, int)> arithmeticFcn { getArithmeticFunction( getOperator() ) };
92      std::cout << arithmeticFcn(getInteger(), getInteger()) << '\n';
93
94      return 0;
95  }
```

nascardriver
December 9, 2019 at 4:39 am · Reply

1.
I wouldn't say so. I'll reply again later.

2.
`arithmeticArray` shouldn't be global, it can be moved into `getArithmeticFunction`.
This quiz might get removed entirely, I'll leave it as-is for now.

3.
You can store the return values in variables before passing them to `arithmeticFcn`

```
1  int x{ getInteger() };
2  int y{ getInteger() };
3  std::cout << arithmeticFcn(x, y) << '\n';
```

nascardriver

December 11, 2019 at 3:55 am · Reply

I talked to Alex, the lesson has been updated. Back to your comments,

1.
No matter if you use regular function pointers or std::function, you should use a type alias if the type is used in multiple places, which it is.
This quiz doesn't require the use of `std::function`, but it doesn't hurt using it :)

2.
The quiz has been removed, because it didn't follow good practice and didn't add challenges concerning the current chapter.

Ryan
December 7, 2019 at 12:56 pm · Reply

Why can't non reference parameter treated as the same as referenced parameter for function pointers?? Is it the reasons that were discussed in the passing arguments lessons.

```
1  int foo(int &x) // reference
2  {
3      return x;
4  }
5
6  int main()
7  {
8      int (*fcnPtr)(int&){ foo }; //okay
9      int (*fcnPtr)(int) { foo }; //error
10
11     int value{ 5 }; // l-value for reference
12     std::cout << fcnPtr(value) << '\n';
13
14     return 0;
15 }
```

nascardriver
December 8, 2019 at 5:19 am · Reply

The type of the function pointer has to match that of the function. `int&` is not the same as `int`.

Ged
December 6, 2019 at 1:24 pm · Reply

Sorry for posting a second time, but forgot to add one more question.

1. Since you talked about typedef I remembered that there was a huge data type that I had no clue what it meant.

```
1  typedef std::vector<std::pair<std::string, int> > pairlist_t; // make pairlist_t an alias fo
2  pairlist_t pairlist; // instantiate a pairlist_t
```

Because you talked about this in the previous chapters I understand what it means. But the part "std::pair<std::string, int>" confuses me a bit. You only showed this function if we wanted to return multiple values from a function. But how does it work with an array(vector)? How to we declare it? How to take lets say the 5th index of the vector int variable. Is this even used?

Ged
December 6, 2019 at 1:43 pm · Reply

I just found the reply button I'm always used to it being at the bottom that didn't even notice it at the top :D

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

int main()
{
    std::vector<std::pair<std::string, int>> vector(5);
    using index_t = std::vector<std::pair<std::string, int>>::size_type;
    for (index_t index{ 0 }; index < vector.size(); ++index)
    {
        std::cout << std::get<0>(vector[index]) << std::get<1>(vector[index]) << '\n'
    }
    return 0;
}
```

This code will print out the whole vector. But if the user needs to input every element, how do we write that? Or lets say we want to initialise it like (int array[2] {2,4}). How should the {2,4} look like in our vector?

nascardriver
December 7, 2019 at 4:34 am · Reply

`std::pair` is a type on its own, the vector doesn't really matter.

```
#include <string>
#include <utility> // std::pair

int main() {
  std::pair<std::string, int> p{"hello", 123};

  p = {"wow", 13};

  // std::pair has @first and @second, std::tuple doesn't.
  p.first = "mew";
  p.second = 9;

  return 0;
}
```

When you add the vector, you have a list of these pairs. Each element has a string and an int.

```
#include <string>
#include <utility>
#include <vector>

int main() {
  std::vector<std::pair<std::string, int>> v{
      {"hello", 123},
      {"cat", 1},
      {"bye", 321}
  };

  v[0] = {"wow", 13};
  v[1].first = "mew";
```

```
14    v[2].second = 9;
15
16    return 0;
17  }
```

Ged
December 6, 2019 at 11:47 am · Reply

1.
Shouldn't (size) be (size - 1) because we can't compare the last element with anything? This actually doesn't change anything because the 2nd for will be equal to size and never be used, but adds an extra cycle that we don't need.

```
1   for (int startIndex = 0; startIndex < size; ++startIndex)
```

2. Unfortunately, type inference won't work for function parameters (even if they have default values), so its use is somewhat limited.

I didn't understand this sentence. You used

```
1  auto fcnPtr = foo;
2  std::cout << fcnPtr(5);
```

And the function had a parameter so how does it not work?

nascardriver
December 7, 2019 at 4:26 am · Reply

1.
You're right. The last iteration does nothing. The inner loop won't run and the last element will be swapped with itself. Lesson updated, thanks!

2.
I don't understand that sentence (Or paragraph, a type alias also doesn't make return type/parameters visible) either, @Alex.

Alex
December 10, 2019 at 8:04 pm · Reply

re #2: was intended to be a restatement of the fact that you can't use "auto" as a function parameter type. I removed the sentence, as it's not a problem specific to function pointers.

You're correct that type aliases hide type/parameters too. That is a shared downside of both type inference and type aliases -- that said, they're still worth using appropriately.

Tompouce
November 26, 2019 at 2:05 pm · Reply

Hi! I'm having a perplexing problem over here °~°
I'm encountering an issue in my getInt() function to request an int from the user, here's the code:

```
1  int getInt()
2  {
3      std::cout << "Enter an integer:\n";
4      int inputInt;
```

```
5       do
6       {
7           std::cout << "> ";
8           std::cin.clear();
9           bool fail {std::cin.fail()};
10          std::cin >> inputInt;
11          fail = std::cin.fail();
12          std::cin.ignore(maxStreamSize, '\n');
13      } while (std::cin.fail());
14      return inputInt;
15  }
```

When the while loop is entered the user is prompted as normal. But when I give an input that would put cin in a fail state -like entering 'h'- cin fails, inputInt is set to 0, then it loops back as expected.
But then, eventhough std::cin.clear() does reset the failure state of cin (I watched the fail variable in a debugger) the next std::cin >> inputInt; line does not prompt the user, and cin goes back to a failure state.
This leads to an endless loop.
The only thing that comes to my mind that could cause this would be if there was erroneous input left in the cin buffer, but cin.ignore() should take care of that... (maxStreamSize is a global constexpr set to numeric_limits<streamsize>::max())

So yeah I don't get it :/ And it's not the first time I've used similar code to get user input, yet this is the first time I've had this issue.

edit:
I've changed the function to

```
1   int getInt()
2   {
3       std::cout << "Enter an integer:\n";
4       int inputInt;
5       while ((std::cout << "> ") && !(std::cin >> inputInt))
6       {
7           std::cin.clear();
8           std::cin.ignore(maxStreamSize, '\n');
9       }
10      return inputInt;
11  }
```

and now it works just fine, but I still don't know why the previous code didn't work...

nascardriver
November 27, 2019 at 4:13 am · Reply

In your first snippet
Line 10 fails, the stream enters a failed state.
Line 12 doesn't do anything, because the stream is in a failed state.
Line 8 clears the error flag, but the bad input is still in the stream.
Line 10 fails again, repeat.

In the second snippet, you're clearing the error flag before calling `ignore`, so that's working fine.

Tompouce
November 27, 2019 at 9:02 am · Reply

Oooh okay, I assumed cin.ignore would work even in a fail state
Thanks for clearing that out!

« Older Comments 1 ... 4 5 6