25.5 — Early binding and late binding

In this lesson and the next, we are going to take a closer look at how virtual functions are implemented. While this information is not strictly necessary to effectively use virtual functions, it is interesting. Nevertheless, you can consider both sections optional reading.

When a C++ program is executed, it executes sequentially, beginning at the top of main(). When a function call is encountered, the point of execution jumps to the beginning of the function being called. How does the CPU know to do this?

When a program is compiled, the compiler converts each statement in your C++ program into one or more lines of machine language. Each line of machine language is given its own unique sequential address. This is no different for functions -- when a function is encountered, it is converted into machine language and given the next available address. Thus, each function ends up with a unique address.

Binding refers to the process that is used to convert identifiers (such as variable and function names) into addresses. Although binding is used for both variables and functions, in this lesson we're going to focus on function binding.

Early binding

Most of the function calls the compiler encounters will be direct function calls. A direct function call is a statement that directly calls a function. For example:

```
#include <iostream>

void printValue(int value)
{
    std::cout << value;
}

int main()
{
    printValue(5); // This is a direct function call
    return 0;
}
```

Direct function calls can be resolved using a process known as early binding. **Early binding** (also called static binding) means the compiler (or linker) is able to directly associate the identifier name (such as a function or variable name) with a machine address. Remember

that all functions have a unique address. So when the compiler (or linker) encounters a function call, it replaces the function call with a machine language instruction that tells the CPU to jump to the address of the function.

Let’s take a look at a simple calculator program that uses early binding:

```
#include <iostream>

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 main()
{
    int x{};
    std::cout << "Enter a number: ";
    std::cin >> x;

    int y{};
    std::cout << "Enter another number: ";
    std::cin >> y;

    int op{};
    std::cout << "Enter an operation (0=add, 1=subtract, 2=multiply): ";
    std::cin >> op;

    int result {};
    switch (op)
    {
        // call the target function directly using early binding
        case 0: result = add(x, y); break;
        case 1: result = subtract(x, y); break;
        case 2: result = multiply(x, y); break;
        default:
            std::cout << "Invalid operator\n";
            return 1;
    }

    std::cout << "The answer is: " << result << '\n';

    return 0;
}
```

Because add(), subtract(), and multiply() are all direct function calls, the compiler will use early binding to resolve the add(), subtract(), and multiply() function calls. The compiler will replace the add() function call with an instruction that tells the CPU to jump to the address of

the add() function. The same holds true for subtract() and multiply().

Late Binding

In some programs, the function being called can't be resolved until runtime. In C++, this is sometimes known as **late binding** (or in the case of virtual function resolution, **dynamic binding**).

Author's note

In general programming terminology, the term "late binding" means the function being called is looked up by name at runtime. C++ does not support this. In C++, the term "late binding" is typically used in cases where the actual function being called is not known by the compiler or linker at the point where the function call is actually being made. Instead, the function to be called has been determined (at runtime) somewhere prior to that point.

In C++, one way to get late binding is to use function pointers. To review function pointers briefly, a function pointer is a type of pointer that points to a function instead of a variable. The function that a function pointer points to can be called by using the function call operator `()` on the pointer.

For example, the following code calls the add() function:

```
#include <iostream>

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

int main()
{
    // Create a function pointer and make it point to the add function
    int (*pFcn)(int, int) { add };
    std::cout << pFcn(5, 3) << '\n'; // add 5 + 3

    return 0;
}
```

Calling a function via a function pointer is also known as an indirect function call. The following calculator program is functionally identical to the calculator example above, except it uses a function pointer instead of a direct function call:

```
#include <iostream>

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 main()
{
    int x{};
    std::cout << "Enter a number: ";
    std::cin >> x;

    int y{};
    std::cout << "Enter another number: ";
    std::cin >> y;

    int op{};
    std::cout << "Enter an operation (0=add, 1=subtract, 2=multiply): ";
    std::cin >> op;

    // Create a function pointer named pFcn (yes, the syntax is ugly)
    int (*pFcn)(int, int) { nullptr };

    // Set pFcn to point to the function the user chose
    switch (op)
    {
        case 0: pFcn = add; break;
        case 1: pFcn = subtract; break;
        case 2: pFcn = multiply; break;
        default:
            std::cout << "Invalid operator\n";
            return 1;
    }

    // Call the function that pFcn is pointing to with x and y as parameters
    // This uses late binding
    std::cout << "The answer is: " << pFcn(x, y) << '\n';

    return 0;
}
```

In this example, instead of calling the add(), subtract(), or multiply() function directly, we've instead set pFcn to point at the function we wish to call. Then we call the function through the pointer. The compiler is unable to use early binding to resolve the function call `pFcn(x, y)` because it can not tell which function pFcn will be pointing to at compile time!

Late binding is slightly less efficient since it involves an extra level of indirection. With early binding, the CPU can jump directly to the function's address. With late binding, the program has to read the address held in the pointer and then jump to that address. This involves one extra step, making it slightly slower. However, the advantage of late binding is that it is more flexible than early binding, because decisions about what function to call do not need to be made until run time.

In the next lesson, we'll take a look at how late binding is used to implement virtual functions.