25.5 — Early binding and late binding

ALEX[1] OCTOBER 20, 2024

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 and dispatching

Our programs contain many names (identifiers, keywords, etc…). Each name has a set of associated properties: for example, if the name represents a variable, that variable has a type, a value, a memory address, etc…

For example, when we say `int x`, we're telling the compiler to associate the name `x` with the type `int`. Later if we say `x = 5`, the compiler can use this association to type check the assignment to ensure it is valid.

In general programming, **binding** is the process of associating names with such properties. **Function binding** (or **method binding**) is the process that determines what function definition is associated with a function call. The process of actually invoking a bound function is called **dispatching**.

In C++, the term binding is used more casually (and dispatching is usually considered part of binding). We'll explore the C++ use of the terms below.

Nomenclature

Binding is an overloaded term. In other contexts, binding may refer to:

- The binding of a reference to an object
- `std::bind`
- Language 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:

```cpp
#include <iostream>

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

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

int main()
{
    printValue(5);   // direct function call to printValue(int)

    Foo f{};
    f.printValue(5); // direct function call to Foo::printValue(int)
    return 0;
}
```

In C++, when a direct call is made to a non-member function or a non-virtual member function, the compiler can determine which function definition should be matched to the call. This is sometimes called **early binding** (or **static binding**), as it can be performed at compile-time. The compiler (or linker) can then generate machine language instructions that tells the CPU to jump directly to the address of the function.

For advanced readers

If we look at the assembly code generated for the call to `printValue(5)` (using clang x86-64), we see something like this:

```
        mov     edi, 5              ; copy argument 5 into edi register in prepar
        call    printValue(int)     ; directly call printValue(int)
```

You can clearly see that this is a direct function call to printValue(int).

Calls to overloaded functions and function templates can also be resolved at compile-time:

```cpp
#include <iostream>

template <typename T>
void printValue(T value)
{
    std::cout << value << '\n';
}

void printValue(double value)
{
    std::cout << value << '\n';
}

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

int main()
{
    printValue(5);   // direct function call to printValue(int)
    printValue<>(5); // direct function call to printValue<int>(int)

    return 0;
}
```

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

```cpp
#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 to non-member functions, the compiler will match these function calls to their respective function definitions at compile-time.

Note that because of the switch statement, which function is actually called is not determined until runtime. However, that is a path of execution issue, not a binding issue.

Late binding

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

Author's note

In general programming terminology, the term "late binding" usually means that the function being called can't be determined based on static type information alone, but must be resolved using dynamic type information.

In C++, the term tends to be used more loosely to mean any function call 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.

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 `printValue()` function through a function pointer:

```
#include <iostream>

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

int main()
{
    auto fcn { printValue }; // create a function pointer and make it point to
function printValue
    fcn(5);                  // invoke printValue indirectly through the function
pointer

    return 0;
}
```

Calling a function via a function pointer is also known as an indirect function call. At the point where `fcn(5)` is actually called, the compiler does not know at compile-time what function is being called. Instead, at runtime, an indirect function call is made to whatever function exists at the address held by the function pointer.

For advanced readers

If we look at the assembly code generated for the call to `fcn(5)` (using clang x86-64), we see something like this:

```
        lea     rax, [rip + printValue(int)] ; determine address of printValue
        mov     qword ptr [rbp - 8], rax     ; move value in rax register into

        mov     edi, 5                       ; copy argument 5 into edi regist
        call    qword ptr [rbp - 8]          ; invoke the function at the addr
```

You can clearly see that this is a indirect function call to printValue(int) via its address.

The following calculator program is functionally identical to the calculator example above, except it uses a function pointer instead of a direct function call:

```cpp
#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;

    using FcnPtr = int (*)(int, int); // alias ugly function pointer type
    FcnPtr fcn { nullptr }; // create a function pointer object, set to nullptr initially

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

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

    return 0;
}
```

In this example, instead of calling the `add()`, `subtract()`, or `multiply()` function directly, we've instead set `fcn` 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 `fcn(x, y)` because it can not tell which function `fcn` 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 runtime.

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

Next lesson
25.6 The virtual table

2

Back to table of contents

3

Previous lesson
25.4 Virtual destructors, virtual assignment, and overriding virtualization

4

5

131 COMMENTS Newest

Steven
December 6, 2024 3:34 am PST

Early binding equivalent to static binding.
Late binding equivalent to dynamic dispatch. Isn't it a little bit weird?

0 Reply

Filipe

December 4, 2024 1:27 pm PST

I still don't understand why some virtual functions calls can't be resolved at compile time, since if write:

Base* a = &derived;

outside any conditional statement, than by this line of code the compiler already knows which object is assigned to the Base pointer 'a'. And since we knows the vtable of the derived he knows which function has to call, directly call.

And even for what i understand it should be able to perform a quicker indirect call since the compiler would know in advance the address of the derived vtable and the offset of the function in that address.

There's something i'm missing me here.

Last edited 7 months ago by Filipe

1 Reply

John

Reply to Filipe [7] March 28, 2025 1:54 am PDT

The only thing I am seeing as a potential issue/reason is that 'Base* a' could potentially be re-assigned to point to a different type of it's derived/child. And being that code is evaluated from top to bottom, it does make sense why, unless it was able to go back after realization that there's no subsequent reassignment.

0 Reply

Alex Author

Reply to Filipe [7] December 6, 2024 1:06 am PST

AFAIK, they can be in certain cases, using an optimization technique known as "devirtualization". https://quuxplusone.github.io/blog/2021/02/15/devirtualization/ is a pretty good article about this.

1 Reply

wangzhichao

July 26, 2024 8:02 am PDT

The last example cannot compile successfully.
I have corrected it as follows:

```cpp
#include <iostream>

// a simple calculator program that uses late binding

using FcnPtr = int (*)(int, int);

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;

    FcnPtr fcn { nullptr };
    switch (op)
    {
    case 0: fcn = add; break;
    case 1: fcn = subtract; break;
    case 2: fcn = multiply; break;
    default: std::cout << "Invalid operator\n"; return 1;
    }

    std::cout << "The answer is: " << fcn(x, y) << '\n';

    return 0;
}
```

1 Reply

Alex Author

Reply to wangzhichao 8 July 26, 2024 4:20 pm PDT

Whoops. Integrated. Thank you!

0 Reply

ftkhateeb

July 22, 2024 4:16 am PDT

Dynamic binding = dynamic polymorphism ?

0 Reply

Swaminathan
July 13, 2024 11:25 pm PDT

In your code below, why should the compiler wait till runtime to find the function addresses? It can already allocate three addresses for those three function definitions(one for add, sub, multiply each). So, what's stopping it from mapping each cases to their corresponding addresses at the compile time itself?

```
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;
}
```

Last edited 11 months ago by Swaminathan

0 Reply

Swaminathan R
Reply to Swaminathan 9 July 15, 2024 10:28 pm PDT

I think I understand after some reflection. Correct me if I am wrong please:

```
std::cout << "The answer is: " << pFcn(x, y) << '\n';
```

Here, cout only knows the memory address of the pFcn object and not the memory address of add, sub, multiply function...Any value could be stored inside(not necessarily add, sub or mul but something else too like some address of max, min functions).. so it leaves it to the run time to look at the value stored, and then treat that value as a function and use arguments a,b to execute that function..

```
std::cout << "The answer is: " << pFcn(x, y) << '\n';
```

0 Reply

Alex Author
Reply to Swaminathan R 10 July 17, 2024 1:35 pm PDT

Yup, this. `pFcn` can point to any valid function, and which function is currently being pointed to can be changed at runtime. So which function is actually called is determined (at runtime) based on the address held by `pFcn`.

0 Reply

Swaminathan R

Reply to Alex 11 July 17, 2024 10:43 pm PDT

Thank you!! One more question

int* ptr {};
……some code………
std::cout<<*ptr;

Here, Because ptr may have stored any int address at runtime when the cout executes, the compiler may not know at compile time which address to print the cout with. So I assume this also late binding?

Last edited 11 months ago by Swaminathan R

0 Reply

Alex Author

Reply to Swaminathan R 12 July 18, 2024 7:57 pm PDT

No, because `*ptr` doesn't invoke a function. This is just printing a value at runtime. There is no binding involved.

0 Reply

Swaminathan

Reply to Alex 13 July 18, 2024 9:20 pm PDT

Sorry if I am being pushy. But it seems the same to me in the case of std::cout<<pFcn(a,b) as well. What do we bind here?

the only extra step in pFcn(a,b) is like you said, it has to execute the function to evaluate to an int. So what exactly makes it a late binding whereas *ptr an early binding?

Last edited 11 months ago by Swaminathan

0 Reply

Alex Author

Reply to Swaminathan 14 July 20, 2024 6:46 pm PDT

I didn't say it was an early binding. :)

In C++ the terms early binding and late binding are used a bit differently than in the general programming sense. In the context of C++ function calls, early binding means the compiler can directly resolve the call at compile-time, whereas late binding means the call must be resolved at runtime. AFAIK, the term binding isn't used to talk about things like values and types.

0 Reply

Strain
April 27, 2024 2:19 pm PDT

Wouldn't inputting an invalid operator be an error? Then it might be better to return 1.

...right?

0 Reply

Alex Author
Reply to Strain 15 April 28, 2024 10:07 am PDT

Sure, though it probably doesn't matter since this program is unlikely to be invoked by another program that needs to know whether it succeed or not. Examples updated.

0 Reply

Strain
March 29, 2024 9:23 am PDT

Rather than doing a `do while`, I would use a `while (true)` with a `break` in the end and a `continue` on the `switch`'s `default`.

0 Reply

Alex Author
Reply to Strain 16 March 30, 2024 12:14 pm PDT

Yeah, that's better, as it removes the redundant check on the value of `op`. That said, I went a slightly simpler direction instead and removed the loop altogether, since it's not relevant to the point of the example.

0 Reply

learnccp lesson reviewer
July 25, 2023 3:13 pm PDT

```
unsigned int prng() {
  unsigned int x =
std::chrono::high_resolution_clock::now().time_since_epoch().count();
  x ^= (x << 13);
  x ^= (x >> 17);
  x ^= (x << 5);
  return x & 0x7FFFFFFF;
}

//x86_64:

//prng():
//        sub     rsp, 8 base stack pointer pointing 8 places below, -8 bytes.
//        call    std::chrono::_V2::system_clock::now() calling clock
//        mov     rdx, rax value of rax MOVED onto rdx
//        sal     eax, 13 shift arithmetic right 13 places
//        xor     eax, edx Logical Exclusive OR between these 2, result stored into
eax
//        mov     edx, eax eax value MOVED onto edx
//        shr     edx, 17 shift right 17 bits
//        xor     edx, eax  Logical Exclusive OR between these 2, result stored into
edx
//        mov     eax, edx value of edx moved into eax
//        sal     eax, 5 shift arithmetic right eax by 5 places.
//        xor     eax, edx  Logical Exclusive OR between these 2, result stored into
eax
//        and     eax, 2147483647 logical AND between 21... and eax, result goes into
eax
//        add     rsp, 8 moves base pointer back to 0 in the stack
//        ret(urn)
```

0 Reply

Strain

Reply to learnccp lesson reviewer [17] March 29, 2024 9:25 am PDT

Why does it loses its greatest bit?

0 Reply

learnccp lesson reviewer

Reply to learnccp lesson reviewer [17] July 25, 2023 3:13 pm PDT

learnasm (https://www.youtube.com/watch?v=_sSFtJwgVYQ)[18]

0 Reply

Strain

Reply to learnccp lesson reviewer [19] April 27, 2024 2:21 pm PDT

I mean, you chose to make it lose that bit. Why did you decide to make it lesser than `2^31` rather than natural `2^32`? Is it for compatibility with `signed int`?

0 Reply

learnccp lesson reviewer
July 25, 2023 3:06 pm PDT

for seeing the assembly code just go to godbolt and paste code. it isnt difficult. all you have to learn are 10 registers 'keywords' as all program basically uses the same registers:

mov, add, ret, pop, push, sub, imul, lea, call, j..., and that's it basically.

0 Reply

learnccp lesson reviewer
July 25, 2023 3:04 pm PDT

Exquisite lesson

1 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/the-virtual-table/
3. https://www.learncpp.com/
4. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/
5. https://www.learncpp.com/early-binding-and-late-binding/
6. https://gravatar.com/
7. https://www.learncpp.com/cpp-tutorial/early-binding-and-late-binding/#comment-604878
8. https://www.learncpp.com/cpp-tutorial/early-binding-and-late-binding/#comment-600097
9. https://www.learncpp.com/cpp-tutorial/early-binding-and-late-binding/#comment-599577
10. https://www.learncpp.com/cpp-tutorial/early-binding-and-late-binding/#comment-599640
11. https://www.learncpp.com/cpp-tutorial/early-binding-and-late-binding/#comment-599716
12. https://www.learncpp.com/cpp-tutorial/early-binding-and-late-binding/#comment-599757
13. https://www.learncpp.com/cpp-tutorial/early-binding-and-late-binding/#comment-599811
14. https://www.learncpp.com/cpp-tutorial/early-binding-and-late-binding/#comment-599822
15. https://www.learncpp.com/cpp-tutorial/early-binding-and-late-binding/#comment-596336
16. https://www.learncpp.com/cpp-tutorial/early-binding-and-late-binding/#comment-595220
17. https://www.learncpp.com/cpp-tutorial/early-binding-and-late-binding/#comment-584496
18. https://www.youtube.com/watch?v=_sSFtJwgVYQ
19. https://www.learncpp.com/cpp-tutorial/early-binding-and-late-binding/#comment-584497
20. https://g.ezoic.net/privacy/learncpp.com