12.4 — Early binding and late binding

BY ALEX ON FEBRUARY 7TH, 2008 | LAST MODIFIED BY ALEX ON DECEMBER 28TH, 2015

In this chapter 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 machine language address.

Binding refers to the process that is used to convert identifiers (such as variable and function names) into machine language 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 nValue)
{
    std::cout << nValue;
}

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 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 machine address. So when the compiler 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>
using namespace std;

int Add(int nX, int nY)
{
    return nX + nY;
}

int Subtract(int nX, int nY)
{
    return nX - nY;
}

int Multiply(int nX, int nY)
{
    return nX * nY;
}
```

```
18
19  int main()
20  {
21      int nX;
22      cout << "Enter a number: ";
23      cin >> nX;
24
25      int nY;
26      cout << "Enter another number: ";
27      cin >> nY;
28
29      int nOperation;
30      do
31      {
32          cout << "Enter an operation (0=add, 1=subtract, 2=multiply): ";
33          cin >> nOperation;
34      } while (nOperation < 0 || nOperation > 2);
35
36      int nResult = 0;
37      switch (nOperation)
38      {
39          case 0: nResult = Add(nX, nY); break;
40          case 1: nResult = Subtract(nX, nY); break;
41          case 2: nResult = Multiply(nX, nY); break;
42      }
43
44      cout << "The answer is: " << nResult << endl;
45
46      return 0;
47  }
```

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, it is not possible to know which function will be called until runtime (when the program is run). This is known as **late binding** (or dynamic binding). 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:

```
1   int Add(int nX, int nY)
2   {
3       return nX + nY;
4   }
5
6   int main()
7   {
8       // Create a function pointer and make it point to the Add function
9       int (*pFcn)(int, int) = Add;
10      cout << pFcn(5, 3) << endl; // add 5 + 3
11
12      return 0;
13  }
```

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:

```
1   #include <iostream>
2   using namespace std;
3
4   int Add(int nX, int nY)
5   {
```

```
6       return nX + nY;
7   }
8
9   int Subtract(int nX, int nY)
10  {
11      return nX - nY;
12  }
13
14  int Multiply(int nX, int nY)
15  {
16      return nX * nY;
17  }
18
19  int main()
20  {
21      int nX;
22      cout << "Enter a number: ";
23      cin >> nX;
24
25      int nY;
26      cout << "Enter another number: ";
27      cin >> nY;
28
29      int nOperation;
30      do
31      {
32          cout << "Enter an operation (0=add, 1=subtract, 2=multiply): ";
33          cin >> nOperation;
34      } while (nOperation < 0 || nOperation > 2);
35
36      // Create a function pointer named pFcn (yes, the syntax is ugly)
37      int (*pFcn)(int, int);
38
39      // Set pFcn to point to the function the user chose
40      switch (nOperation)
41      {
42          case 0: pFcn = Add; break;
43          case 1: pFcn = Subtract; break;
44          case 2: pFcn = Multiply; break;
45      }
46
47      // Call the function that pFcn is pointing to with nX and nY as parameters
48      cout << "The answer is: " << pFcn(nX, nY) << endl;
49
50      return 0;
51  }
```

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(nX, nY)` 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 compiler can tell the CPU to 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.

12.5 -- The virtual table

Index

12.3 -- Virtual destructors, virtual assignment, and overriding virtualization

Share this:

C++ TUTORIAL | PRINT THIS POST

38 comments to 12.4 — Early binding and late binding

Arijit Chattopadhyay
July 29, 2008 at 4:24 am · Reply

Good example.. Easy to understand

sssss
August 26, 2008 at 4:20 am · Reply

very good example

RajenKumar
March 23, 2010 at 9:20 am · Reply

Vary good example….

gswrg
October 23, 2008 at 1:25 am · Reply

Why do you use the “&” in the first example