Jhf5my - John Fishbein Thursday, April 11, 2019

CS 2150 Post-Lab Report

Dynamic Dispatch

Dynamic Dispatch is the situation in which it is unclear at compile time which class member function should be invoked in terms of inheritance. When this occurs in c++, if the virtual keyword is present in the method header, the decision is made at run time, otherwise the compiler makes a decision at compile time. When the virtual keyword is present, the run-time decision is made using a virtual method table. Essentially, when the object is instantiated the address of the desired method is placed into the virtual method table. A pointer is then added to the object which points to the table, allowing it to be accessed. This allows the proper method to be called at runtime. Consider this example code below: If the virtual keyword were not in front of func, the compiler would not know which version of func to go to when var->func is invoked. Because it would not know, the c++ compiler would default to the func method in class A. When executed without the virtual keyword, this code prints 0. But, the virtual keyword is present. This causes a virtual method table to be created when var is instantiated. Because of this table, the compiler is able to figure out which version of func to go to and therefore would execute the func in class B. When the virtual keyword is present, this code prints 10.

```
class A {
public:
  int x;
  A(){ x =0;}
  A(int n){
    x = n;
  }

  virtual int func(){
    return x;
  }
};
class B : public A {
public:
  int y;
  B(){y = 0;}
  B(int n){
    y = n;
  }

  int func(){
    return 2*y;
  }
};
```

```
int main(){
  A* var = new B(5);

  cout << var->func() << endl;


  return 0;
}
```

Optimized Code

Using the -O2 flag during compilation tells the compiler to attempt to optimize the code. It does this by attempting both to reduce the number of instructions needed to achieve the desired output, and by reducing the amount of memory used in the computation. If the compiler deems a set of instructions to be unnecessary or inefficient, it will remove those instructions or attempt to find a faster way to do the same operation. For example, consider this simple but inefficient implementation of a sum function.

```
int mySum(int a, int b, int c){
  int z = a;
  z += b;
  z += c;

  return z;

  for (int i = 0; i < 10; i++){
    cout << "THIS CODE IS UNREACHABLE" << endl;
  }

}
```

Without the -O2 flag, that code compiles to the code shown in Figure A. However, when the -O2 flag is used, the code compiles to the clearly more efficient version shown in figure B.

```
_Z5mySumiii:                            # @_Z5mySumiii
        .cfi_startproc
# %bb.0:
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset rbp, -16
        mov     rbp, rsp
        .cfi_def_cfa_register rbp
        mov     dword ptr [rbp - 4], edi
        mov     dword ptr [rbp - 8], esi
        mov     dword ptr [rbp - 12], edx
        mov     edx, dword ptr [rbp - 4]
        mov     dword ptr [rbp - 16], edx
        mov     edx, dword ptr [rbp - 8]
        add     edx, dword ptr [rbp - 16]
        mov     dword ptr [rbp - 16], edx
        mov     edx, dword ptr [rbp - 12]
        add     edx, dword ptr [rbp - 16]
        mov     dword ptr [rbp - 16], edx
        mov     eax, dword ptr [rbp - 16]
        pop     rbp
        ret
```

```
_Z5mySumiii:
        .cfi_startproc
# %bb.0:

        lea     eax, [rdi + rsi]
        add     eax, edx
        ret
```

The non-optimized version of the assembly code contains many inefficiencies. First, it unnecessarily uses the stack. Accessing the stack takes much longer than simply

reading and writing to registers, so when possible, the -O2 flag will eliminate stack usage. Another inefficiency comes when the code compounds the sum into a local variable. The -O2 flag causes the compiler to remove the use of the unnecessary local variable and does the calculation much faster. Clearly, in certain cases the -O2 flag greatly optimizes code.