John Zheng
jz5pt
4/11/19
inlab9

Optimization

For the inlab, I chose to research the topic of optimization using the -O2 flag. I've included the sample code I wrote. The first function, helloWorld, just prints "Hello World!" to the console. The second function, positive, takes in an integer and returns true if the integer is positive. I call the functions in the main method. When I compile the code into assembly without the -O2 flag, the first thing I noticed was how long it was. It was 1582 lines just for the C++ code below. After I put the -O2 flag in, the assembly code was reduced to just 584 lines, still a lot but a big improvement. I've included the assembly code for the positive function below, the left one is the assembly with optimization. You can see the noticeable difference between the two. Without the optimization, the commands are really basic and there is a lot of mov commands. The optimized function uses the test command, which is essentially doing a bitwise AND and compare at the same time.

```
1  using namespace std;
2
3  #include <string>
4  #include <iostream>
5
6  void helloWorld(){
7      cout << "Hello World!" << endl;
8  }
9
10 bool positive(int x) {
11     return x > 0;
12 }
13
14 int main() {
15     helloWorld();
16     positive(7);
17
18     return 0;
19 }
```

```
118     .globl  __Z8positivei
119     .p2align    4, 0x90
120 __Z8positivei:
121     .cfi_startproc
122 ## %bb.0:
123     push    rbp
124     .cfi_def_cfa_offset 16
125     .cfi_offset rbp, -16
126     mov rbp, rsp
127     .cfi_def_cfa_register rbp
128     test    edi, edi
129     setg    al
130     pop rbp
131     ret
132     .cfi_endproc
133
```

```
76      .p2align    4, 0x90         ## -
    basic_ostreamIT_T0_EES7_
77  __ZNSt3__1L4endlIcNS_11char_traitsIc
    char_traitsIcEEEERNS_13basic_ostream
78      .cfi_startproc
79  ## %bb.0:
80      push    rbp
81      .cfi_def_cfa_offset 16
82      .cfi_offset rbp, -16
83      mov rbp, rsp
84      .cfi_def_cfa_register rbp
85      sub rsp, 32
86      mov qword ptr [rbp - 8], rdi
87      mov rdi, qword ptr [rbp - 8]
88      mov rax, qword ptr [rbp - 8]
89      mov rcx, qword ptr [rax]
90      mov rcx, qword ptr [rcx - 24]
91      add rax, rcx
92      mov qword ptr [rbp - 16], rdi ##
93      mov rdi, rax
94      mov esi, 10
95      call    __ZNKSt3__19basic_iosIcN
96      mov rdi, qword ptr [rbp - 16] ##
97      movsx   esi, al
98      call    __ZNSt3__113basic_ostrea
99      mov rdi, qword ptr [rbp - 8]
100     mov qword ptr [rbp - 24], rax ##
101     call    __ZNSt3__113basic_ostrea
102     mov rcx, qword ptr [rbp - 8]
103     mov qword ptr [rbp - 32], rax ##
104     mov rax, rcx
105     add rsp, 32
106     pop rbp
107     ret
108     .cfi_endproc
109
```

Below is the code snippet for a product function that uses loops. Next to it are the code snippets in assembly, optimized in the middle. In this case, the compiler seems to know what the function's purpose is. Instead of using mov and conditional jumps, it just uses integer multiply on the first and the second parameter.

```
14 int multiply(int x, int y) {
15     int sum = 0;
16     for (int i = 0; i < y; i++) {
17         sum += x;
18     }
19     return sum;
20 }
```

```
136 __Z8multiplyii:
137     .cfi_startproc
138 ## %bb.0:
139     push    rbp
140     .cfi_def_cfa_offset 16
141     .cfi_offset rbp, -16
142     mov rbp, rsp
143     .cfi_def_cfa_register rbp
144     imul    edi, esi
145     xor eax, eax
146     test    esi, esi
147     cmovg   eax, edi
148     pop rbp
149     ret
150     .cfi_endproc
151
```

```
131 __Z8multiplyii:
132     .cfi_startproc
133 ## %bb.0:
134     push    rbp
135     .cfi_def_cfa_offset 16
136     .cfi_offset rbp, -16
137     mov rbp, rsp
138     .cfi_def_cfa_register rbp
139     mov dword ptr [rbp - 4], edi
140     mov dword ptr [rbp - 8], esi
141     mov dword ptr [rbp - 12], 0
142     mov dword ptr [rbp - 16], 0
143 LBB5_1:
144     mov eax, dword ptr [rbp - 16]
145     cmp eax, dword ptr [rbp - 8]
146     jge LBB5_4
147 ## %bb.2:
148     mov eax, dword ptr [rbp - 4]
149     add eax, dword ptr [rbp - 12]
150     mov dword ptr [rbp - 12], eax
151 ## %bb.3:
152     mov eax, dword ptr [rbp - 16]
153     add eax, 1
154     mov dword ptr [rbp - 16], eax
155     jmp LBB5_1
156 LBB5_4:
157     mov eax, dword ptr [rbp - 12]
158     pop rbp
159     ret
160     .cfi_endproc
161
```

Overall, the main take away from this is that the non-optimized assembly has a lot of redundancy and the optimized assembly reduces redundancy by using more advance commands.

Dynamic Dispatch

After researching this topic, I've concluded the following key concepts: dynamic dispatch is a key component of object-oriented programming language, with dynamic dispatch the compiler makes the decision of which method to run at compile time, and it adds a lot of flexibility to the program by using virtual functions. Below I've written a sample code that includes class vegetable and class spinach which inherits vegetable. I wrote two virtual functions which are then overridden by the spinach class. The functions simply just print whether it is a spinach or vegetable. In the main function I created vegetable pointer which points to a spinach object, thus the program prints the spinach's functions.

```
1  #include <iostream>
2  using namespace std;
3
4  class vegetable {
5      public:
6          virtual void printVegetable() {
7              cout << "This is a vegetable" << endl;
8          }
9          virtual void vegetableLoop() {
10             for (int i = 0; i < 3; i++) {
11                 cout << "I am a vegetable" << endl;
12             }
13         }
14 };
15
16 class spinach: public vegetable {
17     public:
18         void printVegetable() {
19             cout << "This is a spinach" << endl;
20         }
21         void vegetableLoop() {
22             for (int i = 0; i < 3; i++) {
23                 cout << "I am a spinach" << endl;
24             }
25         }
26 };
27
28 int main() {
29     vegetable *veggy;
30     spinach s;
31     veggy = &s;
32     veggy->printVegetable();
33     veggy->vegetableLoop();
34
35
36     return 0;
37 }
```

```
This is a spinach
I am a spinach
I am a spinach
I am a spinach
Johns-MacBook-Pro-5:postlab9 johnzheng$
```

Below is part of the assembly code generated. It seems in the main function that the only thing being called is from the spinach class. Nowhere does it show anything from the vegetable class. This makes sense since the veggy pointer points to the spinach object.

```
 6 _main:
 7     .cfi_startproc
 8 ## %bb.0:
 9     push    rbp
10     .cfi_def_cfa_offset 16
11     .cfi_offset rbp, -16
12     mov rbp, rsp
13     .cfi_def_cfa_register rbp
14     sub rsp, 32
15     mov dword ptr [rbp - 4], 0
16     lea rdi, [rbp - 24]
17     call    __ZN7spinachC1Ev
18     lea rdi, [rbp - 24]
19     mov qword ptr [rbp - 16], rdi
20     mov rdi, qword ptr [rbp - 16]
21     mov rax, qword ptr [rdi]
22     call    qword ptr [rax]
23     mov rax, qword ptr [rbp - 16]
24     mov rdi, qword ptr [rax]
25     mov qword ptr [rbp - 32], rdi
26     mov rdi, rax
27     mov rax, qword ptr [rbp - 32]
28     call    qword ptr [rax + 8]
29     xor eax, eax
30     add rsp, 32
31     pop rbp
32     ret
33     .cfi_endproc
34
```

Work cited

https://lukasatkinson.de/2016/dynamic-vs-static-dispatch/
https://www.geeksforgeeks.org/virtual-function-cpp/