Matt Calayo mjc5gh 4/11/18 postlab9.pdf

Dynamic Dispatch

Dynamic dispatch is the process by which different implementation of methods are used in runtime. This is done in c++ by using the virtual keyword. To show how this works I made a base class named feline and two subclasses named cat and lion. Each of these classes have three functions: sound, purr, and eat. In the feline base class, sound and eat are both declared with the virtual keyword. In the cat and lion subclasses I have different implementations for each. Last, in the main method I create an object and have the object call the sound purr and eat methods. Since the sound and eat methods are both virtual and use dynamic dispatch, they call the version of sound and eat defined in the subclass. This results in the output of “meow, purrpurr, and nibble” for the cat class function calls. The way that these methods are called in assembly is interesting. For the statically dispatched methods, assembly first moves the parameters into the right registers and then calls the function of the feline base class. The way that assembly deals with dynamic dispatch is different however. Instead of directly calling the function, multiple dereferences are made to access the correct function. The first operation is moving a dereferenced base pointer -8 to rax, then dereferencing rax twice, then moving a dereferenced base pointer -8 to rdx then to rdi then calling rax. This works because a pointer to this function is stored at this location.

```
class feline
{
public:
  virtual void sound(){
    cout<<"hello"<<endl;
  }
  void purr(){
    cout<<"purrpurr"<<endl;
  }
  virtual void eat(){
    cout<<"nomnom"<<endl;
  }
};
```

```
class cat:public feline
{
public:
  void sound(){
    cout<<"meow"<<endl;
  }
  void purr(){
    cout<<"purrpurrpurr"<<endl;
  }
  void eat(){
    cout<<"nibble"<<endl;
  }
};
```

```
mov rax, QWORD PTR [rbp-8]
mov rdi, rax
call feline::purr()
```

```
mov rax, QWORD PTR [rbp-8]
mov rax, QWORD PTR [rax]
mov rax, QWORD PTR [rax]
mov rdx, QWORD PTR [rbp-8]
mov rdi, rdx
call rax
```

The dereferences of rax allow the assembly code to call rax and it is actually calling the function sound. In the assembly code, because the purr function of feline is not declared as virtual, there is only one implementation of purr. Even though cat and lion have implementations of purr, since they cannot be called they are not even included in the assembly code. Since the sound and eat functions are virtual, there are two different implementations of each function, one for cat and one for lion, but not one for feline.

When the code is created each type of class has a vtable created for virtual functions. The vtable for each class shows which functions are virtual which helps decide whether to use static or dynamic dispatch.

```
vtable for lion:
  .quad 0
  .quad typeinfo for lion
  .quad lion::sound()
  .quad lion::eat()
```

Templates

For the simple templated class that I wrote, I wrote a function that finds the minimum of two values. This function was designed to find the minimum of any different type, as a template does. The actual defining of the function looks like any normal function. The difference in the assembly code was when I call the function using two different types. I called the function on two ints and two longs. In the assembly code, two entirely different functions were defined. These functions were minimum<int> and minimum<long> as shown on the right. There is not much different for these two functions. The only apparent difference is the locations that are being accessed by the pointers. If I only call one version of the function, the actual function looks the same but there is only one. The code looks very similar because the function that assembly uses to compare numbers, cmp, seems like it is able to function with different types and no changes to the syntax. In the main method each function is called respectively for each type. Vectors use templates in the same way as this. They are able to use any time given to them to perform functions. For each different type that is called when using the template the assembly code creates a new function. This allows for every type to be used but also makes sure that not too much code is created so that not too much space is used.

```
int minimum<int>(int, int):
  push rbp
  mov rbp, rsp
  mov DWORD PTR [rbp-20], edi
  mov DWORD PTR [rbp-24], esi
  mov eax, DWORD PTR [rbp-20]
  cmp eax, DWORD PTR [rbp-24]
  jge .L4
  mov eax, DWORD PTR [rbp-20]
  jmp .L5
.L4:
  mov eax, DWORD PTR [rbp-24]
.L5:
  mov DWORD PTR [rbp-4], eax
  mov eax, DWORD PTR [rbp-4]
  pop rbp
  ret
long minimum<long>(long, long):
  push rbp
  mov rbp, rsp
  mov QWORD PTR [rbp-24], rdi
  mov QWORD PTR [rbp-32], rsi
  mov rax, QWORD PTR [rbp-24]
  cmp rax, QWORD PTR [rbp-32]
  jge .L8
  mov rax, QWORD PTR [rbp-24]
  jmp .L9
.L8:
  mov rax, QWORD PTR [rbp-32]
.L9:
  mov QWORD PTR [rbp-8], rax
  mov rax, QWORD PTR [rbp-8]
  pop rbp
  ret
```

```
template <class tType>
tType minimum( tType x, tType y){
  tType ret;
  ret = (x<y)? x:y;
  return ret;
}
```

```
int main(){
  int a=10,b=5,c;
  long x=20,y=50,z;
  c=minimum<int>(a,b);
  z=minimum<long>(x,y);
```