C++ Analysis

Chapter 20
OOP

- Functions (i.e. methods) in C++ associated with particular classes of objects
  - Classes used to define objects -> object: instance of class, same method
  - Similar to struct, but also include functions
- “this” pointer
  - Implicit pointer to object that holds the variable being accessed
  - (By default) Passed as a compiler-generated parameter to a function (typically the ECX register, sometimes ESI)
  - Listing 20-2, Listing 20-3, p. 430
    - Loads this pointer into ecx
    - puts ecx into eax, then access x to compare it to 10
class SimpleClass {
public:
    int x;
    void HelloWorld() {
        if (x == 10) printf("X is 10.\n");
    }
    ...
};

int _tmain(int argc, _TCHAR* argv[]) {
    SimpleClass myObject;
    myObject.x = 9;
    myObject>HelloWorld();
    SimpleClass myOtherObject;
    myOtherObject.x = 10;
    myOtherObject>HelloWorld();
}

;Main Function
push    ebp
mov     ebp, esp
sub     esp, 1F0h
mov     [ebp+var_10], offset 404768
mov     [ebp+var_C], 9
lea     ecx, [ebp+var_10]
call    sub_4115D0
mov     [ebp+var_34], offset 404768
mov     [ebp+var_30], 0Ah
lea     ecx, [ebp+var_34]
call    sub_4115D0

;HelloWorld Function
push    ebp
mov     ebp, esp
sub     esp, 9Ch
push    ebx
push    esi
push    edi
mov     [ebp+var_4], ecx
mov     eax, [ebp+var_4]
cmp     dword ptr [eax+4], 0Ah
jnz     short loc_4115F6
push    offset axIs10_ "X is 10.\n"
call    ds:_imp_printf

Listing 20-3: The this pointer shown in disassembly
Overloading and Mangling

- Method overloading allows multiple functions to have same name, but accept different parameters.
- When function called, compiler determines which version to use (according to parameters).
  - C++ uses name mangling to support this construct in the PE file.
  - Algorithm for mangling is compiler-specific.
    - IDA Pro demangles based on what it knows about specific compilers.
      - ?TestFunction@SimpleClass@@QAEXHH@Z
      - public: void __thiscall SimpleClass::TestFunction(int,int)
    - Shows the original function name and parameters.
    - IDAPro supports Microsoft, Borland, Watcom, Visual Age, GNU
- Inheritance is not visible in assembly code (a feature, does not require any runtime data structure.)
Virtual vs. Nonvirtual Functions

- Virtual functions
  - Has the same name defined in child class
  - **Can be overridden by a child class (polymorphism)**
  - Execution is determined at runtime with the child class overriding the parent
  - Can keep parent functionality by changing the type of the object to be an instance of the parent class
- Example: parent class `Socket` with a virtual function called `sendData`, two child classes `UDPSocket` and `TCPSocket` to override `sendData` function with specific protocol
Virtual vs. Nonvirtual Functions

- Nonvirtual functions
  - Execution is determined at compile time
  - If object is an instance of the parent, the parent class's function will be called, even if the object at run-time belongs to the child class, see example at Table 20-1, Page 433.
Defined as Class A
Determined at Compile time

Non-virtual function

class A {
    public:
        void foo() {
            printf("Class A\n");
        }
};

class B : public A {
    public:
        void foo() {
            printf("Class B\n");
        }
};

```c
void g(A& arg) {
    arg.foo();
}
```

int _tmain(int argc, _TCHAR* argv[]) {
    B b;
    A a;
    g(b);
    return 0;
}

Virtual function

class A {
    public:
        virtual void foo() {
            printf("Class A\n");
        }
};

class B : public A {
    public:
        virtual void foo() {
            printf("Class B\n");
        }
};

```c
void g(A& arg) {
    arg.foo();
}
```

int _tmain(int argc, _TCHAR* argv[]) {
    B b;
    A a;
    g(b);
    return 0;
}

Defined as Class A
Determined runtime
Virtual Function Tables

- C++ compiler add special data structure to support virtual function tables
- Code look the same – Assembly Different
- Non-virtual call vs. Virtual call

<table>
<thead>
<tr>
<th>Non-virtual function call</th>
<th>Virtual function call</th>
</tr>
</thead>
<tbody>
<tr>
<td>00401000 push ebp</td>
<td>00401000 push ebp</td>
</tr>
<tr>
<td>00401001 mov ebp, esp</td>
<td>00401001 mov ebp, esp</td>
</tr>
<tr>
<td>00401003 mov ecx, [ebp+arg_0]</td>
<td>00401003 mov eax, [ebp+arg_0]</td>
</tr>
<tr>
<td><strong>00401006 call sub_401030</strong></td>
<td><strong>00401006 mov edx, [eax]</strong></td>
</tr>
<tr>
<td>0040100B pop ebp</td>
<td>00401008 mov ecx, [ebp+arg_0]</td>
</tr>
<tr>
<td>0040100C retn</td>
<td>0040100B mov eax, [edx]</td>
</tr>
<tr>
<td></td>
<td><strong>0040100D call eax</strong></td>
</tr>
<tr>
<td></td>
<td>0040100F pop ebp</td>
</tr>
<tr>
<td></td>
<td>00401010 retn</td>
</tr>
</tbody>
</table>

 Argument – a reference to function
V Table

- Each class (with virtual function) has own vtable
- Vtable -> first 4 bytes of the object

- The first 4-byte entry of the vtable is a pointer to the code for the first virtual function (next 4 bytes -> first function)
- See which offset is being called to figure out which function is called
Recognize VTable

- Usually after `new`...
- Only the first value has cross-reference, the rest -> offsets

```
004020F0  off_4020F0  dd offset sub_4010A0
004020F4  dd offset sub_4010C0
004020F8  dd offset sub_4010E0
```
Creating and Destroying Objects

- **Constructor and Destructor functions**
  - Constructor is called when object is created; destructor is called when object is destroyed
  - Object either stored on stack for local variables
  - Object stored in heap if “new” is used (Listing 20-8, p. 437) -> keywords for creating heap space
    - Initializes vtable for object stored on stack
    - Does multiple loads of vtable (parent, then child)
    - Creates another object via “new” call (see the name mangling)