1 Inheritance

Consider the following example of simple inheritance in c++. It includes a derived class "Child" that inherits from the base class "Parent." It inherits Parent's x and y members and also has a public member of its own, "luckyFin."

```
class Parent
{
   public:
      void setX(int w)
      {
         x = w;
      }
      void setY(int h)
      {
         y = h;
      }
   protected:
      int x;
      int y;
};

class Child: public Parent
{
   public:
      int xPlusY()
      {
         return (x + y);
      }

      int luckyFin;
};

int main(void)
{
   Child * nemo = new Child();

   nemo->setX(2);
   nemo->setY(3);

   int add = nemo->xPlusY();

   return 0;
}
```

The resulting assembly code is too long to list in full but snippets will be shown as the topics they are relevant to are discussed.

1.1 Initialization

The following x86 snippet shows the initialization of the nemo object.

```
    call  _Znwj
    add   esp, 16
    mov   DWORD PTR [eax], 0
    mov   DWORD PTR [eax+4], 0
    mov   DWORD PTR [eax+8], 0
    mov   DWORD PTR [ebp-12], eax
    mov   eax, DWORD PTR [ebp-12]
```

The call to _Znwj returns a pointer to the location of the newly created nemo object in memory. Then the data members of nemo are moved into memory at that location.

1.2 Data Layout

From the code above it is not immediately clear which element is stored at which location. This information can be deduced by considering a call to setX inherited from the Parent class. The assembly for such a call is as follows.

```
    push  2
    push  eax
    call  _ZN6Parent4setXEi
```

with the function itself being

```
_ZN6Parent4setXEi:
    push  ebp
    mov   ebp, esp
    mov   eax, DWORD PTR [ebp+8]
    mov   edx, DWORD PTR [ebp+12]
    mov   DWORD PTR [eax], edx
    pop   ebp
    ret
```

Before the call is made, eax holds the address of the nemo object in memory. The setX method then moves the argument '2' located at [ebp+12] into [eax], the lowest memory location in nemo. Similar analysis of the setY function shows us that y is located at [eax+4]. From this we can assume that 'luckyFin' is stored at [eax+8].

The general rule that we can conclude from this examination is that within an object that inherits from a parent class the object's own member variables are stored first (at higher memory addresses) and then the inherited members are stored in reverse order with the first member of the parent class being stored at the lowest address.

1.3 Destruction

The first example will demonstrate what happens when a user created object goes out of scope. In order to more easily demonstrate this the c++ code has been modified in the following ways:

```
void scope() {
    Child * nemo = new Child();
    nemo->setX(2);
    nemo->setY(3);
    int add = nemo->xPlusY();
}

int main(void) {
    scope();
    return 0;
}
```

The last lines of assembly within the 'scope' function call look like this:

```
   call  _ZN5Child6xPlusYEv
   add   esp, 16
   mov   DWORD PTR [ebp-16], eax
   leave
   ret
```

As can be seen, the deallocation of the nemo object is very simple and operates just like deallocation of any other variables on the stack by adding to esp.

The second example will show what happens when something is 'destroyed' by the destructor.

```
class Planet {
   public:
      void setX(int w) {
         x = w;
      }
      void setY(int h) {
         y = h;
      }
   protected:
      int x;
      int y;
};

class DevourerOfWorlds {
   public:
      int luckyFin;
      DevourerOfWorlds();
      ~DevourerOfWorlds();
```

```

    private:
      Planet * earth;
};

DevourerOfWorlds::DevourerOfWorlds() {
    earth = new Planet();
}

DevourerOfWorlds::~DevourerOfWorlds() {
    delete earth;
}

void scope() {
    DevourerOfWorlds galactus = DevourerOfWorlds();
}

int main(void) {
    scope();
    return 0;
}
```

Here the constructors and destructors have been explicitly defined. The destructor is automatically called when the program exits the scope of scope(). They compile to

```
_ZN16DevourerOfWorldsC2Ev:
   push ebp
   mov  ebp, esp
   sub  esp, 8
   sub  esp, 12
   push 8
   call _Znwj
   add  esp, 16
   mov  DWORD PTR [eax], 0
   mov  DWORD PTR [eax+4], 0
   mov  edx, DWORD PTR [ebp+8]
   mov  DWORD PTR [edx+4], eax
   leave
   ret
_ZN16DevourerOfWorldsD2Ev:
   push ebp
   mov  ebp, esp
   sub  esp, 8
   mov  eax, DWORD PTR [ebp+8]
   mov  eax, DWORD PTR [eax+4]
   sub  esp, 12
   push eax
   call _ZdlPv
```

```
    add  esp, 16
    leave
    ret
```

The constructor initializes galactus as we expect and also calls _Znwj to create the Planet earth object.

In the destructor, the address of earth is moved into eax which is then passed as an argument to _ZdlPv which deallocates the memory allocated to it. From there things proceed as normal with memory being deallocated by adding to esp and then leave and ret making up the standard epilogue.

2 Optimization

The following c++ code will be used to demonstrate a number of examples of different types of optimizations that might be done by a compiler. All optimizations are done using the -O2 flag.

```
int constantArithmetic() {
    int x = 0;
    for (int i = 0; i < 100; i++) {
        x += i;
    }
    return x;
}

int redundancy(int x) {
    int y = 2 * x;
    int z = 2 * x;
    return y + z;
}

int simplification(int x) {
    int y = x + x;
    int z = 2 * x;
    int ret = y - z;
    return ret;
}

int inlining(int x) {
    return x + 1;
}

int main(void) {
    int arg;
    cin >> arg;
    int x = constantArithmetic();
    int y = redundancy(arg);
    int z = simplification(arg);
```

```
    int a = inlining(arg);
    cout << a;
    return 0;
}
```

2.1 Constant Arithmetic

Compare the unoptimized version of the function

```
_Z18constantArithmeticv:
   push ebp
   mov  ebp, esp
   sub  esp, 16
   mov  DWORD PTR [ebp-4], 0
   mov  DWORD PTR [ebp-8], 0
   mov  eax, DWORD PTR [ebp-8]
   add  DWORD PTR [ebp-4], eax
   add  DWORD PTR [ebp-8], 1
   cmp  DWORD PTR [ebp-8], 99
   mov  eax, DWORD PTR [ebp-4]
   leave
   ret
```

to the optimized version.

```
_Z18constantArithmeticv:
   mov  eax, 4950
   ret
```

In the unoptimized version the code loops naively as we expect. The optimized version demonstrates the compiler's ability to do constant arithmetic at compile time. This function does not depend on any user input or variable quanitities and will always return the same value. The compiler recognizes this and is able to compute the value, even though the calculation is done within a loop and isn't entirely trivial, and insert the value directly into the code.

2.2 Redundancy

Compare the unoptimized version of the function

```
_Z10redundancyi:
   push ebp
   mov  ebp, esp
   sub  esp, 24
   mov  eax, DWORD PTR [ebp+8]
   add  eax, eax
   mov  DWORD PTR [ebp-12], eax
   mov  eax, DWORD PTR [ebp+8]
```

```
9   add  eax, eax
10  mov  DWORD PTR [ebp-16], eax
11  sub  esp, 8
12  push DWORD PTR [ebp-12]
13  push OFFSET FLAT:_ZSt4cout
14  call _ZNSolsEi
15  add  esp, 16
16  sub  esp, 8
17  push DWORD PTR [ebp-16]
18  push OFFSET FLAT:_ZSt4cout
19  call _ZNSolsEi
20  add  esp, 16
21  leave
22  ret
```

to the optimized version.

```
_Z10redundancyi:
  push ebx
  sub  esp, 16
  mov  eax, DWORD PTR [esp+24]
  lea  ebx, [eax+eax]
  push ebx
  push OFFSET FLAT:_ZSt4cout
  call _ZNSolsEi
  pop  eax
  pop  edx
  push ebx
  push OFFSET FLAT:_ZSt4cout
  call _ZNSolsEi
  add  esp, 24
  pop  ebx
  ret
```

The code in both cases is fairly long and involves strange function calls in order to print things to stdout. The optimization can be seen in lines 6-9 of the unoptimized code. It computes the values of z and z seperately and then loads and prints each one individually. In the optimized version the compiler recognizes that y and z will have the same value. Thus is calculates the value once and then prints it twice, avoiding a redundant computation.

2.3 Simplification

Compare the unoptimized version of the function

```
_Z14simplificationi:
  push ebp
  mov  ebp, esp
  sub  esp, 16
```

```
    mov   eax, DWORD PTR [ebp+8]
    add   eax, eax
    mov   DWORD PTR [ebp-4], eax
    mov   eax, DWORD PTR [ebp+8]
    add   eax, eax
    mov   DWORD PTR [ebp-8], eax
    mov   eax, DWORD PTR [ebp-4]
    sub   eax, DWORD PTR [ebp-8]
    mov   DWORD PTR [ebp-12], eax
    mov   eax, DWORD PTR [ebp-12]
    leave
    ret
```

to the optimized version.

```
_Z14simplificationi:
    xor   eax, eax
    ret
```

The optimized version of the code recognizes that x-x must equal - and simply places that value in eax. This is different from constant arithmetic because the value of x could be anything at runtime. The compiler is demonstrating awareness of basic mathematical axioms in order to simplify the result even when it knows nothing about the input.

2.4 Inlining

The difference here takes place not in the functions implementation but in its calling. In the unoptimized version of the code it is called as

```
    push  eax
    call  _Z8inliningi
```

In the optimized version no call is ever made to the function. The result of the function is displayed to the user so the compiler can't choose to just ignore it. Instead, recognizing that the function is trivially simple, it decided to inline the code into main. It looks like this.

```
    add   eax, 1
```

In addition to the differences mentioned above there were some optimizations that took place in every function. First, the standard prologue and epilogue are ignored in the optimized version. Second, because the results of the simplification and constant arithmetic functions calls go unused neither of them were ever actually called in the optimized code.

References

I found the following links useful in trying to understand my assembly code.

1. http://en.wikipedia.org/wiki/Loop_unrolling
2. http://en.wikipedia.org/wiki/Redundant_Code
3. http://en.wikipedia.org/wiki/Category:Compiler_optimizations
4. http://stackoverflow.com/questions/5474355/about-leave-in-x86-assembly
5. http://www.compileroptimizations.com/category/expression_simplification.htm