STATIC ANALYSIS II

Paolo Palumbo

F-Secure Corporation

Initial considerations

- During this lecture we will focus on compiled languages, specifically C/C++
  - The compiler used for the examples will be Visual C/C++ compiler as included in Visual Studio Express 2010
- The same techniques can be applied to different languages and compilers

TO THERE AND BACK!

Creating a program

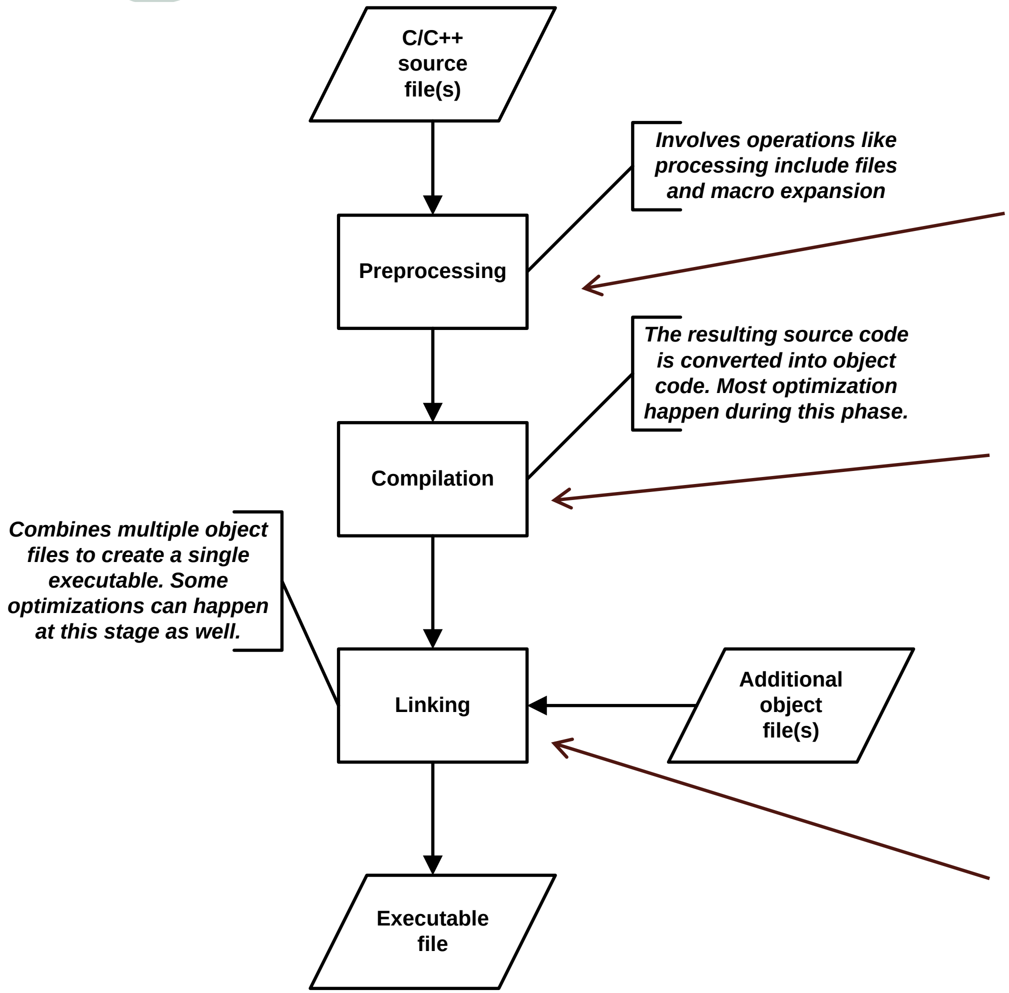
A great idea is born!

The idea is expressed using a programming language suitable to a human

Program is compiled and linked

An object suitable to machine understanding is created. In the end, this is all about ones and zeros ☺

Simplified compilation and linking process for C/C++



No more macros or preprocessor definitions

Comments are discarded, non-needed symbol names as well. Structured data types cease to exist. Code is mangled and transformed according to optimizations and translation process

Additional modules are statically linked to the object file corresponding to the program

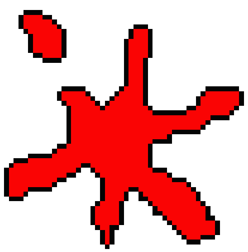
Reverse Code Engineering

- Deals with the opposite of the process that we saw before
  - For interpreted languages, we still need to undo what the bytecode compiler has done
- The ultimate goal is not the rebuilding of the original source code
  - The original source code cannot be recovered, but equivalent source can
    - It is a very lengthy and complicated process
- Usually, knowledge of the program's inner workings is what is needed
  - For example, when performing malware analysis, the researcher wants to get an understanding of what the malware does

Tools to aid the binary Reverse Code Engineering process

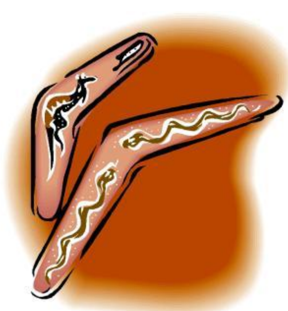
Disassemblers: translate the machine code into the equivalent human readable assembler representation. Some frequently used disassemblers:

- IDA Pro
- HIEW
- HT-Editor
- PE Browse Professional
- …



Debuggers: step through the code as the processor executes it. Examples are:

- Ollydbg/Immunity Debugger
- IDA Pro
- Windbg
- …



Decompilers: translate the machine code into high-level language source code. The process of decompilation is extremely complex, and most of the available tools are not able handle real-world programs automatically. Examples are:

- Hex-Rays decompiler
- Boomerang
- REC Studio
- …

THE USUAL SUSPECT

The great program™

Program1.cpp

(Global Scope)

```cpp
// Program1.cpp : Defines the entry point for the console
//

#include "stdafx.h"

#define SUPER_CONSTANT 3
typedef int MY_SUPERIOR_DATA_TYPE;

int foobar(int x, int y)
{
    MY_SUPERIOR_DATA_TYPE z = x + y + SUPER_CONSTANT;
    return z;
}

int _tmain(int argc, _TCHAR* argv[])
{
    MY_SUPERIOR_DATA_TYPE z = foobar(1, 2);
    printf("Result is %i! :)\n", z);
    return 0;
}
```

Perfect example of a great idea turned into code!

Inside the great program™

```
.text:0040132E ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
.text:0040132E
.text:0040132E ; Attributes: library function
.text:0040132E
.text:0040132E                 public wmainCRTStartup
.text:0040132E wmainCRTStartup proc near
.text:0040132E                 call    __security_init_cookie
.text:00401333                 jmp     __tmainCRTStartup
.text:00401333 wmainCRTStartup endp
.text:00401333

.text:004010A4 ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
.text:004010A4
.text:004010A4 ; Attributes: bp-based frame
.text:004010A4
.text:004010A4 __tmainCRTStartup proc near             ; CODE XREF: wmainCRTStartup+5↓j
.text:004010A4
.text:004010A4 var_1C          = dword ptr -1Ch
.text:004010A4 ms_exc          = CPPEH_RECORD ptr -18h
.text:004010A4
.text:004010A4                 push    10h
.text:004010A6                 push    offset unk_4021F8
.text:004010AB                 call    __SEH_prolog4
.text:004010B0                 xor     ebx, ebx
.text:004010B2                 cmp     _NoHeapEnableTerminationOnCorruption, ebx
.text:004010B8                 jnz     short loc_4010C5
.text:004010BA                 push    ebx
.text:004010BB                 push    ebx
.text:004010BC                 push    1
.text:004010BE                 push    ebx
.text:004010BF                 call    ds:__imp__HeapSetInformation@16 ; HeapSetInformation(x
.text:004010C5
.text:004010C5 loc_4010C5:                             ; CODE XREF: __tmainCRTStartup+14↑j
.text:004010C5                 mov     [ebp+ms_exc.disabled], ebx
.text:004010C8                 mov     eax, large fs:18h
.text:004010CE                 mov     esi, [eax+4]
.text:004010D1                 mov     [ebp+var_1C], ebx
.text:004010D4                 mov     edi, offset __native_startup_lock
.text:004010D9
.text:004010D9 loc_4010D9:                             ; CODE XREF: __tmainCRTStartup+59↓j
.text:004010D9                 push    ebx             ; Comperand
.text:004010DA                 push    esi             ; Exchange
.text:004010DB                 push    edi             ; Destination
.text:004010DC                 call    ds:__imp__InterlockedCompareExchange@12 ; InterlockedC
```

What is this code? We did not have anything like this in our simple program!

It is the C Runtime startup code and it has been inserted by the linker. It provides the basic support for C/C++ runtime. A few of the features of this code:

- initialize the heap
- parse the command line
- more

We are looking at code disassembled by a powerful tool (IDA), and we have symbols. Usually, things are not so nice.

Inside the great program™ - continued

```
.text:004011A2 loc_4011A2:                                  ; CODE XREF: __tmainCRTStartup+E3↑j
.text:004011A2                                              ; __tmainCRTStartup+F2↑j
.text:004011A2                 mov     eax, envp
.text:004011A7                 mov     ecx, ds:_imp____winitenv
.text:004011AD                 mov     [ecx], eax
.text:004011AF                 push    envp
.text:004011B5                 push    argv
.text:004011BB                 push    argc
.text:004011C1                 call    wmain
.text:004011C6                 add     esp, 0Ch
.text:004011C9                 mov     mainret, eax
.text:004011CE                 cmp     managedapp, ebx
.text:004011D4                 jnz     short loc_40120D
.text:004011D6                 push    eax             ; int
.text:004011D7                 call    ds:__imp__exit
```

The invocation of our code happens much later, inside the `__tmainCRTStartup` routine.

The main function receives three arguments that were prepared by the CRT startup code:

- argc – argument count
- argv – array of pointers to incoming arguments
- envp – array of pointers to environmental variables

Inside the great program™ - continued

```
.00401000: 55              push    ebp                      Foobar
.00401001: 8BEC            mov     ebp,esp
.00401003: 51              push    ecx
.00401004: 8B450C          mov     eax,[ebp][00C]
.00401007: 8B4D08          mov     ecx,[ebp][8]
.0040100A: 8D540103        lea     edx,[ecx][eax][3]
.0040100E: 8955FC          mov     [ebp][-4],edx
.00401011: 8B45FC          mov     eax,[ebp][-4]
.00401014: 8BE5            mov     esp,ebp
.00401016: 5D              pop     ebp
.00401017: C3              retn ; _^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_
.00401018: CC              int     3
.00401019: CC              int     3
.0040101A: CC              int     3
.0040101B: CC              int     3
.0040101C: CC              int     3
.0040101D: CC              int     3
.0040101E: CC              int     3
.0040101F: CC              int     3
.00401020: 55              push    ebp                      Main
.00401021: 8BEC            mov     ebp,esp
.00401023: 51              push    ecx
.00401024: 6A02            push    2
.00401026: 6A01            push    1
.00401028: E8D3FFFFFF      call    .000401000 --↑1
.0040102D: 83C408          add     esp,8
.00401030: 8945FC          mov     [ebp][-4],eax
.00401033: 8B45FC          mov     eax,[ebp][-4]
.00401036: 50              push    eax
.00401037: 68EC204000      push    0004020EC ;'Result is %i! :)' --↓2
.0040103C: FF15A0204000    call    printf
.00401042: 83C408          add     esp,8
.00401045: 33C0            xor     eax,eax
.00401047: 8BE5            mov     esp,ebp
.00401049: 5D              pop     ebp
.0040104A: C3              retn ; _^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_
```

- All of our high level constructs are gone! Thank you compiler!
- The code for the foobar subroutine is also different
    - `lea edx, [ecx][eax][3]` ?

- `lea: load effective address`
- `also used by the compiler to perform effective additions and multiplications`
- `could read also as:`
    - `lea edx, [ecx + eax + 3]`
- `edx = ecx + eax + 3` → `performs the addition as in our source program`

TWENTY THOUSANDS LEAGUES UNDER THE SOURCE CODE

Simple control flow statements

Program2.cpp

(Global Scope)

```
#include "stdafx.h"


int _tmain(int argc, _TCHAR* argv[])
{
    int counter1;

    // A simple for loop
    for (counter1 = 0; counter1 < 10; counter1++)
    {
        printf("[FOR LOOP] Iteration #%i\n", counter1);
    }

    // A simple while loop
    int counter2 = 0;
    while(counter2 < 10)
    {
        printf("[WHILE LOOP] Iteration #%i\n", counter2);
        counter2 ++;
    }

    // A simple do-while loop
    int counter3 = 0;
    do
    {
        printf("[DO-WHILE LOOP] Iteration #%i\n", counter3);
        counter3 ++;
    }while(counter3 < 10);

    goto label1;

    printf("[DEAD CODE] I should be skipped!\n");

label1:
    printf("[GOTO] Reached target destination!\n");

    return 0;
}
```

We will use Visual Studio's C/C++ compiler to see what happens to our code when it is compiled

The program has been compiled and linked with all optimizations disabled

Pre-test loops: for and while loops

```c
int counter1;

// A simple for loop
for (counter1 = 0; counter1 < 10; counter1++)
{
    printf("[FOR LOOP] Iteration #%i\n", counter1);
}
```

```
.text:00401006
.text:00401006 @@for_loop:                                 ;
.text:00401006                 mov     [ebp+counter1], 0 ; counter1 = 0
.text:0040100D                 jmp     short @@_for_loop_header
.text:0040100F ; ---------------------------------------------------------------------------
.text:0040100F
.text:0040100F @@for_loop_increment:                       ; CODE XREF: SimpleProgram+30↓j
.text:0040100F                 mov     eax, [ebp+counter1]
.text:00401012                 add     eax, 1
.text:00401015                 mov     [ebp+counter1], eax ; counter = counter + 1
.text:00401018
.text:00401018 @@_for_loop_header:                         ; CODE XREF: SimpleProgram+D↑j
.text:00401018                 cmp     [ebp+counter1], 10
.text:0040101C                 jge     short @@while_loop ; if counter1 >= 10 goto @@while_loop
.text:0040101E
.text:0040101E @@ffor_loop_body:
.text:0040101E                 mov     ecx, [ebp+counter1]
.text:00401021                 push    ecx
.text:00401022                 push    offset aForLoopIterati ; "[FOR LOOP] Iteration #%i\n"
.text:00401027                 call    ds:printf       ; printf("[FOR LOOP] Iteration #%i\n", counter1);
.text:0040102D                 add     esp, 8
.text:00401030                 jmp     short @@for_loop_increment
.text:00401032 ; ---------------------------------------------------------------------------
```

These kinds of loops perform a check on the loop condition before executing the body of the loop; this means that the body of this kind of loop can be executed zero or more times.

A while loop works in a similar way, as it is another type of pre-test loop.

Post-test loops: do-while loops

```c
// A simple do-while loop
int counter3 = 0;
do
{
    printf("[DO-WHILE LOOP] Iteration #%i\n", counter3);
    counter3 ++;
}while(counter3 < 10);
```

```
.text:0040105C
.text:0040105C @@do_while_loop:                        ; CODE XREF: SimpleProgram+3D↑j
.text:0040105C                 mov     [ebp+counter3], 0 ; counter3 = 0
.text:00401063
.text:00401063 @@do_while_loop_body:                   ; CODE XREF: SimpleProgram+82↓j
.text:00401063                 mov     ecx, [ebp+counter3]
.text:00401066                 push    ecx
.text:00401067                 push    offset aDoWhileLoopIte ; "[DO-WHILE LOOP] Iteration #%i\n"
.text:0040106C                 call    ds:printf       ; printf("[DO-WHILE LOOP] Iteration #%i\n", counter3);
.text:00401072                 add     esp, 8
.text:00401075                 mov     edx, [ebp+counter3]
.text:00401078                 add     edx, 1
.text:0040107B                 mov     [ebp+counter3], edx ; counter3 = counter3 + 1
.text:0040107E
.text:0040107E @@do_while_loop_header:
.text:0040107E                 cmp     [ebp+counter3], 10
.text:00401082                 jl      short @@do_while_loop_body ; if counter3 < 10 goto @@do_while_loop_body
```

The check on the loop condition is done after executing the loop body; this means that the body of a do-while loop will be executed at least one time

The goto statement

```c
    goto label1;

    printf("[DEAD CODE] I should be skipped!\n");

label1:
    printf("[GOTO] Reached target destination!\n");
```

```
.text:00401084 @@goto_statement:                          ; goto @@label1
.text:00401084                 jmp     short @@label1
.text:00401086 ; ---------------------------------------------------------------------------
.text:00401086                 jmp     short @@label1
.text:00401088 ; ---------------------------------------------------------------------------
.text:00401088                 push    offset aDeadCodeIShoul ; "[DEAD CODE] I should be skipped!\n"
.text:0040108D                 call    ds:printf        ; This code is never reached
.text:00401093                 add     esp, 4
.text:00401096
.text:00401096 @@label1:                                  ; CODE XREF: SimpleProgram:@@goto_statement↑j
.text:00401096                                            ; SimpleProgram+86↑j
.text:00401096                 push    offset aGotoReachedTar ; "[GOTO] Reached target destination!\n"
.text:0040109B                 call    ds:printf        ; printf("[GOTO] Reached target destination!\n");
.text:004010A1                 add     esp, 4
```

As in our original source code, control is transferred unconditionally to another point in the program. Please note that the dead code would be removed from final compiled program if even minimal optimizations would have been turned on

Standard C arrays

program3.cpp*
(Global Scope)

```
// program3.cpp : Defines the entry point for
//

#include "stdafx.h"

#define ARRAY_SIZE 0xFF
int my_global_array[ARRAY_SIZE];

int _tmain(int argc, _TCHAR* argv[])
{
    int initializer = 3;

    for (int i = 0; i < ARRAY_SIZE; i++)
    {
        my_global_array[i] = initializer;
    }

    return 0;
}
```

Arrays are implemented as a sequence of memory locations of same size and type. Therefore, there is no difference between them and sequences of unrelated items of the same size and type. Only the code that access them can reveal the semantic association

Code that accesses memory areas in an indexed manner could be a good hint that you are dealing with an array

```
.text:00401006                 mov     [ebp+initializer], 3
.text:0040100D                 mov     [ebp+counter], 0
.text:00401014                 jmp     short @@for_loop_header
.text:00401016 ; ---------------------------------------------------------------------------
.text:00401016
.text:00401016 @@for_loop_increment:                   ; CODE XREF: sub_401000+35↓j
.text:00401016                 mov     eax, [ebp+counter]
.text:00401019                 add     eax, 1
.text:0040101C                 mov     [ebp+counter], eax ; counter = counter + 1
.text:0040101F
.text:0040101F @@for_loop_header:                      ; CODE XREF: sub_401000+14↑j
.text:0040101F                 cmp     [ebp+counter], 0FFh
.text:00401026                 jge     short @@function_exit ; if counter >= 0xFF goto @@function_exit
.text:00401028
.text:00401028 @@for_loop_body:
.text:00401028                 mov     ecx, [ebp+counter]
.text:0040102B                 mov     edx, [ebp+initializer] ; edx = initializer
.text:0040102E                 mov     my_global_array[ecx*4], edx ;
.text:0040102E                                         ;
.text:0040102E                                         ; [0x402020 + ecx * 4] = initializer
.text:0040102E                                         ; 0x402020[ecx * 4] = initializer
.text:0040102E                                         ; my_global_array[ecx * 4] = initializer
.text:0040102E                                         ; my_global_array[ecx * sizeof(int)] = initializer
.text:0040102E                                         ; my_global_array[counter * sizeof(int)] = initializer
.text:0040102E                                         ;
.text:0040102E                                         ; ==> Standard C arrays are implemented as contiguous memory areas
.text:00401035                 jmp     short @@for_loop_increment
```

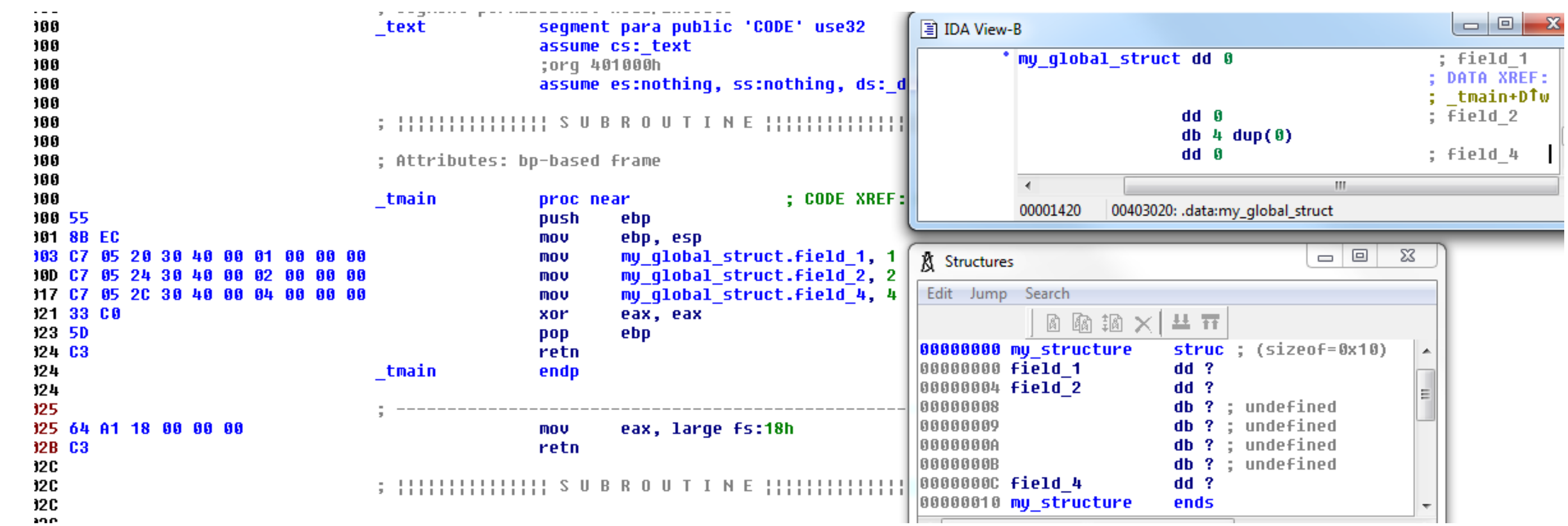
Structures

```
Program4.cpp
_unnamed_struct_0006_1
// Program4.cpp : Defines the entry point
//

#include "stdafx.h"

typedef struct
{
    int field_1;
    int field_2;
    int field_3;
    int field_4;
    int field_5;
}my_struct;

my_struct my_global_struct;

int _tmain(int argc, _TCHAR* argv[])
{
    my_global_struct.field_1 = 1;
    my_global_struct.field_2 = 2;
    my_global_struct.field_4 = 4;
    return 0;
}
```

Similarly to arrays, structures are implemented as a set of contiguos memory locations that contain items of possibly different size and type. The logical association between these elements can only be made by analyzing the code that accesses them

In this example, the type and size of field_3 are unknown, but at least we know that something should be there. Instead, we have no way to know that field_5 is there at all.

Remember about structure alignment when rebuilding structure types!



Unions

For unions, the same memory location is used to store elements of different type. To make this possible, the compiler allocates enough memory to store the biggest item in the union

This makes reversing code that uses unions a bit more challenging, as it may seem initially contradicting.

Program5.cpp

(Global Scope)

```cpp
// Program5.cpp : Defines the entry poin
//

#include "stdafx.h"

typedef union
{
    int my_int;
    char my_char;
}my_union;

my_union my_global_union;

int _tmain(int argc, _TCHAR* argv[])
{
    my_global_union.my_char ='a';
    my_global_union.my_int = 123456;
    return 0;
}
```

```
.text:00401000 _tmain          proc near               ; CODE XREF: start-16D↓p
.text:00401000                 push    ebp
.text:00401001                 mov     ebp, esp
.text:00401003                 mov     my_global_union.my_char, 'a'
.text:0040100A                 mov     my_global_union.my_int, 123456
.text:00401014                 xor     eax, eax
.text:00401016                 pop     ebp
.text:00401017                 retn
.text:00401017 _tmain          endp
.text:00401017
```

Structures

Edit Jump Search

```
00000000
00000000 my_union        union ; (sizeof=0x4)
00000000 my_char         db ?
00000000 my_int          dd ?
00000000 my_union        ends
```

IDA View-B

```
0040301C dword_40301C    dd 44BF19B1h
0040301C
00403020 my_global_union my_union <0>
00403020
00403024 dword_403024    dd 0
```

00001420 00403020: .data:my_global_union

Basics of C++ Classes

Program6.cpp

MySquare

```
// Program6.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"

class MySquare
{
    int side;
    unsigned int id;
public:
    MySquare(int, unsigned int); // Constructor
    int get_area();
    unsigned int get_id();
};

MySquare::MySquare(int input_side, unsigned int input_id)
{
    side = input_side;
    id = input_id;
}

int MySquare::get_area()
{
    return (side * side);
}

unsigned int MySquare::get_id()
{
    return id;
}

int _tmain(int argc, _TCHAR* argv[])
{
    MySquare my_class(10, 1);
    printf("The area of the first square is %i!\n", my_class.get_area());
    printf("The id of the first square is 0x%x\n", my_class.get_id());
    MySquare my_class2(12, 2);
    printf("The area of the second square is %i!\n", my_class2.get_area());
    printf("The id of the second square is 0x%x\n", my_class2.get_id());
    return 0;
}
```

This is avery simple case. No advanced OOP features were used

We have a single class definitions, that provides a simple constructor, a couple of attributes and a two methods. The program then creates two instances of the MySquare class as local variables of the _tmain function.

In this case, after the compilation process, the local variables will contain only instance-specific class members, the attributes.

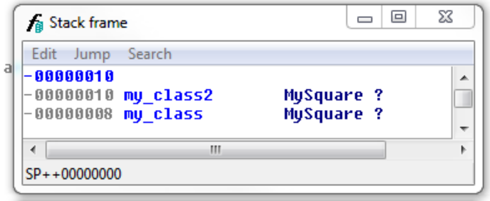
When using additional features of C++, the underlying implementation becomes more complex

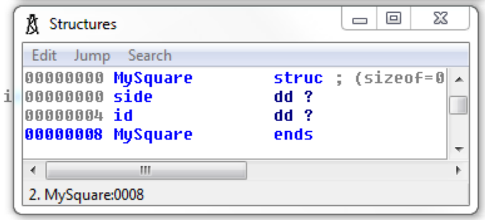
Access specifiers:
- public
- protected
- private

are only constructs designed to help the programmer. After enforcing correctness of the source program, the compiler will remove them and the resulting binary won't have any access specifier

Basics of C++ Classes - continued

```
wmain           proc near               ; CODE XREF: __tmainCRTStartup+11D↓p

my_class2       = MySquare ptr -10h
my_class        = MySquare ptr -8

                push    ebp
                mov     ebp, esp
                sub     esp, 10h        ; On the stack we have space reserved for
                                        ; the attributes of the two class instances
                                        ;
                                        ; The methods are not duplicated!
                push    1
                push    10
                lea     ecx, [ebp+my_class] ;
                                        ; Pass the pointer to the first class instance in the ECX register;
                                        ; other arguments are passed throught the stack.
                                        ;
                                        ; This is the __thiscall convention in action
                                        ;
                call    MySquare__MySquare ; invoke Constructor for the first class instance

MySquare::MySquare(&my_class /* through ECX */, 10, 1);

                lea     ecx, [ebp+my_class]
                call    MySquare__get_area ; Invoke the MySquare::get_area method for the first instance

eax = MySquare::get_area(&my_class);

                push    eax
                push    offset aTheAreaOfTheFi ; "The a
                call    ds:__imp__printf

printf("The area of the first square is %i!\n", eax);

                add     esp, 8
                lea     ecx, [ebp+my_class]
                call    MySquare__get_id

eax = MySquare::get_id(&my_class);

                push    eax
                push    offset aTheIdOfTheFirs ; "The i
                call    ds:__imp__printf

printf("The id of the first square is 0x%x\n", eax);

                add     esp, 8
                push    2
                push    12
                lea     ecx, [ebp+my_class2]
                call    MySquare__MySquare ;
                                        ; Same happens for the second instance. Methods code is
                                        ; reused.
                lea     ecx, [ebp+my_class2]
                call    MySquare__get_area
                push    eax
                push    offset aTheAreaOfTheSe ; "The area of the second square is %i!\n"
                call    ds:__imp__printf
                add     esp, 8
                lea     ecx, [ebp+my_class2]
                call    MySquare__get_id
```






```
MySquare__MySquare proc near            ; CODE XR
                                        ; wmain+4

lpClass         = dword ptr -4
side            = dword ptr  8
id              = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                push    ecx
                mov     [ebp+lpClass], ecx
                mov     eax, [ebp+lpClass]
                mov     ecx, [ebp+side]
                mov     [eax+MySquare.side], ecx
                mov     edx, [ebp+lpClass]
                mov     eax, [ebp+id]
                mov     [edx+MySquare.id], eax
                mov     eax, [ebp+lpClass]
                mov     esp, ebp
                pop     ebp
                retn    8
MySquare__MySquare endp

MySquare__get_area proc near            ; CODE XI
                                        ; wmain+l

lpClass         = dword ptr -4

                push    ebp
                mov     ebp, esp
                push    ecx
                mov     [ebp+lpClass], ecx
                mov     eax, [ebp+lpClass]
                mov     ecx, [ebp+lpClass]
                mov     eax, [eax+MySquare.side]
                imul    eax, [ecx+MySquare.side]
                mov     esp, ebp
                pop     ebp
                retn
MySquare__get_area endp

MySquare__get_id proc near              ; CODE )
                                        ; wmain

lpClass         = dword ptr -4

                push    ebp
                mov     ebp, esp
                push    ecx
                mov     [ebp+lpClass], ecx
                mov     eax, [ebp+lpClass]
                mov     eax, [eax+MySquare.id]
                mov     esp, ebp
                pop     ebp
                retn
MySquare__get_id endp
```

OPTIMIZATION

Constant folding & copy propagation

```
Program9.cpp
(Global Scope)
// Program9.cpp : Defines the entry poin
//

#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
    int x = (12 * 27) +  33;
    int y = x * 2;
    int z = x * y;
    printf("Hello z: %i\n", z);
    return 0;
}
```

Constant folding is responsible for the simplification of constant expressions at compile time:

`x = (12 * 27) + 33 → x = 357`

Copy propagation is responsible for replacing the presence of the target of a direct assignment with its value:

`y = x * 2 → y = 357 * 2`

- Dead code elimination has also been applied here
- These transformations are only possible after dataflow analysis has been performed
- By looking at the final binary, there is no way to know how the source program looked in the first place

```
.text:00401000 ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
.text:00401000
.text:00401000
.text:00401000 wmain           proc near               ; CODE XREF: __tmainCRTStartup+11D↓p
.text:00401000                 push    254898
.text:00401005                 push    offset aHelloZI ; "Hello z: %i\n"
.text:0040100A                 call    ds:__imp__printf
.text:00401010                 add     esp, 8
.text:00401013                 xor     eax, eax
.text:00401015                 retn
.text:00401015 wmain           endp
.text:00401015
```

Dead code elimination

```
Program7.cpp ×
(Global Scope)
// Program7.cpp : Defines the entry point for
//

#include "stdafx.h"


int _tmain(int argc, _TCHAR* argv[])
{
    goto label1;
    printf("I shouldn't be in the code!\n");

label1:
    printf("I should be in the code!\n");

    return 0;
}
```

Dead code elimination is responsible to remove from the final optimized program all of the those parts of the program that the compiler could safely mark as "dead". This includes, for example, unreachable statements . Please note that this optimization will be performed repeatedly during the compilation process

The result of dead code elimination for this sample program is shown below. The line:

```
printf("I shouldn't be in the code!\n");
```

has been removed from the final binary, as there is no execution path that can reach it, and thus it is "dead". As a result of this elimination, the first goto is being eliminated as well, as there is no need for it anymore

```
.text:00401000
.text:00401000 ; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||
.text:00401000
.text:00401000
.text:00401000 wmain          proc near               ; CODE XREF: wmainCRTStartup-126↓p
.text:00401000                push    offset aIShouldBeInThe ; "I should be in the code!\n"
.text:00401005                call    ds:__imp__printf
.text:0040100B                add     esp, 4
.text:0040100E                xor     eax, eax
.text:00401010                retn
.text:00401010 wmain          endp
.text:00401010
```

Inline expansion

Program11.cpp

(Global Scope)

```
// Program11.cpp : Defines the entry point for
//

#include "stdafx.h"

void print_hello(void)
{
    printf("Hello!\n");
}

int _tmain(int argc, _TCHAR* argv[])
{
    for (int i = 0; i < 100000000; i++)
        print_hello();
    return 0;
}
```

Inline expansion consists of replacing the call site of a function with the body of the called function itself. This is done to remove the overhead that comes with the control transfer between caller and callee, plus everything related to the callee's prologue and epilogue code. Inline expansion also opens the door to further optimizations

The most obvious downside is the increase of the code size

```
.text:00401000
.text:00401000 ; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||||
.text:00401000
.text:00401000
.text:00401000 wmain          proc near              ; CODE XREF: __tmainCRTStartup+11D↓p
.text:00401000                push    esi
.text:00401001                mov     esi, ds:__imp__printf
.text:00401007                push    edi
.text:00401008                mov     edi, 100000000
.text:0040100D                lea     ecx, [ecx+0]
.text:00401010
.text:00401010 @@loop:                               ; CODE XREF: wmain+1B↓j
.text:00401010                push    offset aHello   ; "Hello!\n"
.text:00401015                call    esi ; __imp__printf
.text:00401017                add     esp, 4
.text:0040101A                dec     edi
.text:0040101B                jnz     short @@loop
.text:0040101D                pop     edi
.text:0040101E                xor     eax, eax
.text:00401020                pop     esi
.text:00401021                retn
.text:00401021 wmain          endp
.text:00401021
```

There are many more!

- There are many additional optimizations
  - optimizing compilers have been around for decades
    - can turn awful code into something that performs really well
  - a good exercise is to explore additional compiler behavior

- Writing programs in assembly produces faster code?
  - everybody has heard this from someone at some point in their computing career
  - this is rarely the case
    - optimizing compilers can take care of so many things that would be obscure for a human
    - part of the output of the code generator is human-generated anyway
  - sometimes there is the need of handcrafting a special piece of code in assembly to perform a specific task

REAL LIFE EXAMPLE

Simple encryption routine reverse engineering

LIVE

ADDITIONAL READING MATERIAL

Further suggested reading

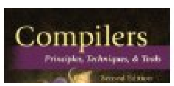
Books › **"compiler design"**

Related Searches: artificial intelligence, compiler.

Showing 1 - 12 of 2,315 Results

Format

Paperback (1,649)	Hardcover (593)	Kindle Edition (42)	HTML (1)

1. 

Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman (Sep 10, 2006)
(11 customer reviews)

OpenRCE: http://www.openrce.org

If you cannot find some particular information, googling helps ☺
http://www.google.com