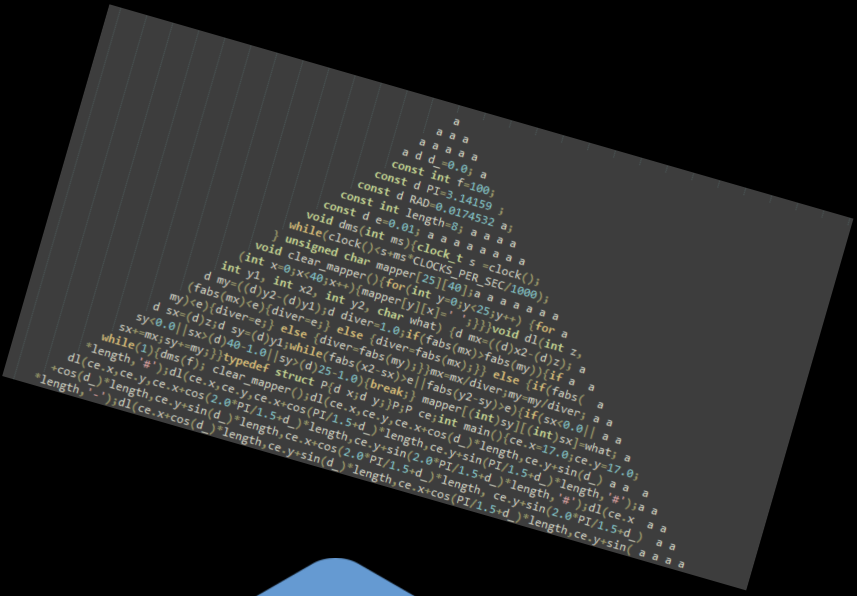


```
                                 a
                               a a a
                              a a a a a
                            a d d_=0.0; a
                          const int f=100;
                         const d PI=3.14159 ;
                       const d RAD=0.0174532 a;
                      const int length=8; a a a a
                     const d e=0.01; a a a a a a a
                  void dms(int ms){clock_t s =clock();
                 while(clock()<s+ms*CLOCKS_PER_SEC/1000);
               } unsigned char mapper[25][40];a a a a a a a a
              void clear_mapper(){for(int y=0;y<25;y++) {for a
            (int x=0;x<40;x++){mapper[y][x]=' ';}}}void dl(int z,
           int y1, int x2, int y2, char what) {d mx=((d)x2-(d)z); a
          d my=((d)y2-(d)y1);d diver=1.0;if(fabs(mx)>fabs(my)){if a a
        (fabs(mx)<e){diver=e;} else {diver=fabs(mx);}} else {if(fabs( a
       my)<e){diver=e;} else {diver=fabs(my);}}mx=mx/diver;my=my/diver; a a
     d sx=(d)z;d sy=(d)y1;while(fabs(x2-sx)>e||fabs(y2-sy)>e){if(sx<0.0|| a a
    sy<0.0||sx>(d)40-1.0||sy>(d)25-1.0){break;} mapper[(int)sy][(int)sx]=what; a
   sx+=mx;sy+=my;}}typedef struct P{d x;d y;}P;P ce;int main(){ce.x=17.0;ce.y=17.0;
  while(1){dms(f); clear_mapper();dl(ce.x,ce.y,ce.x+cos(d_)*length,ce.y+sin(d_) a a a
 *length,'#');dl(ce.x,ce.y,ce.x+cos(PI/1.5+d_)*length,ce.y+sin(PI/1.5+d_)*length,'#');a a
dl(ce.x,ce.y,ce.x+cos(2.0*PI/1.5+d_)*length,ce.y+sin(2.0*PI/1.5+d_)*length,'#');dl(ce.x a a
+cos(d_)*length,ce.y+sin(d_)*length,ce.x+cos(2.0*PI/1.5+d_)*length, ce.y+sin(2.0*PI/1.5+d_) a a
*length,'-');dl(ce.x+cos(d_)*length,ce.y+sin(d_)*length,ce.x+cos(PI/1.5+d_)*length,ce.y+sin( a a a a
```

Introduction to C and C++ obfuscation methods

Covering the basics, for beginners

Gábor Funk
A.K.A Afghan Goat

Table of contents

Beginning

Disclaimer

This writing covers basic and advanced C obfuscation tactics as well as some clever C++ exclusive tactics but most often these tactics can be broken down by a talented reverse engineer. This is a book for beginners and its main goal is to teach people common obfuscation tactics as well a show to reverse engineer them. You will learn about these topics:

- Basic obfuscation tactics using macros
- Obfuscation by obscurity
- Global scope quirks
- Obfuscation where the compiler generates garbage machine code
- Advanced obfuscations offered by the C language
- Control flow inversion
- Obfuscations offered by the C++ language
- How to reverse engineer these obfuscations
- More general awareness about obfuscation
- Deeper knowledge about C and C++ in general

It is also worth noting that this guide assumes you use g++ and gcc for C++ and C code compilation. The snippets can still work in MSVC and other compilers but there is not 100% guarantee.

The C++ code in the examples is optimized for C++17 or later but it is possible that these tactics and snippets will stop working after C++23. I highly doubt that the C code snippets will stop working but currently these snippets compile without an error using g++.

In theory by the time you read this guide, you should have a grasp of basic obfuscation methods and have knowledge of awareness of common obfuscations. Remember, learning to reverse-engineer code is as important as learning to obfuscate your code. Knowing how to decode obfuscated source codes will always be a handy skill and this guide will give you a basic foundation of having that.

Obfuscation is an antipattern

Yes, obfuscating your code involves using a lot of bad practice which makes your code inpossible to understand well without spending extra time for decoding. With this in mind if you plan to obfuscate your program always keep a copy of the source code which is clear from any obfuscation.

No obfuscation is safe

The obfuscations which this guide contains are very simple by nature but even for harder-to-decode obfuscations it only takes an extremely motivated person to crack it. This means that using obfuscation is not viable but you can learn a lot about the different tactics that are out there. You can also utilize this knowledge to decode source files from malicious or untrusted actors.

Prerequisites

Having a very basic C or C++ knowledge is necessary but this book will go through the basics and explain everything in detail (sometimes there will be simplifications for better understanding).

I will also provide Python snippets which automates obfuscation tactics so knowing Python is also a plus but not necessary as the code snippets will be commented.

What is obfuscation and why it is useful?

Obfuscation is the process of deliberately making code, data, or communication difficult to understand or interpret by humans or machines. In the context of software development, obfuscation is commonly used to protect intellectual property, deter reverse engineering, and enhance security by hiding the true intent or functionality of code. Obfuscation makes it harder for attackers or competitors to analyze, copy, or exploit sensitive logic. Obfuscation can consist of using outdated functions or conventions; using weird keywords, types, variable names or inverting the control flow of the functions of your software. All of these features can be called obfuscation.

The practical use of obfuscation is questionable as there are a few things to note:

1. There is no safe, uncrackable obfuscation.
2. Don't rely on obfuscation when trying to hide sensitive data or password. Implement a proper method for that.
3. Insecure or untested obfuscation methods may break your code.

It is also good to know that using strong obfuscation methods may flag your program at some anti-viruses because strong obfuscations are signs of malicious programs. This also helps anti-viruses to prevent false-negatives.

Keep in mind, that obfuscation will have its cost on your software and is not always feasible. When using certain obfuscations (ones which break the control flow or add functionality) you may experience:

- worse performance,
- bigger file size,
- malfunctions

It is your job to decide whenever you need obfuscation or not. **Don't obfuscate your program prematurely!** Also it is worth mentioning once more that no matter how hard you obfuscate your program experienced technicians will still be able to reverse-engineer it. With these out of the way let's get into the basics.

Introduction to pointers

This is a must known important topic when it comes to obfuscation. This is a very simple introduction for using pointers. If you have never met them, don't worry, this section is very simple and is only explaining the basics. A pointer is a variable that stores the memory address of another variable. In most programming languages like C or C++, variables are stored in memory, and each memory location has a unique address. Instead of holding a direct value, a pointer holds this address. Pointers have a type of the value they are pointing to.

To declare a pointer, you use the * symbol:

```
//---CODE

int* ptr;

//---END CODE
```

To assign an address to a pointer, you use the address-of operator „&":

```
//---CODE

#include <stdio.h>

int main() {

    int x=10;

    int* ptr=&x; //Assign the value to the pointer

    printf("value of x: %d",*ptr); /*The *
(dereference operator) extracts the value of the
pointer that it is pointing to.*/

    return 0;

}

//---END CODE
```

To access the value stored at that memory address, you use the dereference operator „*". When printing out the pointer without the dereference operator using the „%p" format we can get the address where the pointer is pointing:

```
//---CODE

#include <stdio.h>

int main() {

    int x=10;

    int* ptr=&x;

    printf("value of ptr: %p",ptr); //No *

    /*Example output: "value of ptr:
0x7fffe44cd324"*/

    return 0;

}

//---END CODE
```

Pointers are simillar to other types: they can be stored in arrays, passed into functions, they can be global variables but it is important to keep in mind that they always point to a value, even when you don't initialize that value. Example of pointer pointing to uninitialized data:

```
//---CODE    int *p;

    printf("value held by p: %d",*p);

    /*Example output: "value held by p: 2055264976"*/

    /*Returns a seemingly random value, this is
called memory junk, it is value left there by other
processes or programs.*/

//---END CODE
```

Pointers also allow functions to modify variables outside their local scope by passing addresses instead of values. Example:

```
//---CODE
```

```
#include <stdio.h>

void add(int* a, int b){ //No return value

    *a+=b;

}

int main() {

    int x=10;

    add(&x,5);

    printf("value of x: %d",x);

    /*Example output: "value of x: 15"*/

    return 0;

}

//---END CODE
```

Memory junk

The previous example showed that an uninitialized pointer can lead to quirky working as it does not give any error or warning (by default). "Memory junk" refers to leftover or uninitialized data present in a memory location. When you allocate memory (for example, using malloc in C) or declare variables without initializing them, that memory might still contain random data from previous operations. This leftover data is often called "garbage values" or "junk data." An example:

```
//---CODE

int x;

printf("%d", x); // x contains junk, unpredictable
value

//---END CODE
```

These values seem random but do not be confused. These values are not pure random and shouldn’t be used as such. Don’t use these values for generating keypairs or anything sensitive as it is not reliable. The values can be predicted (but very hard to predict). This is a common misconception about these values I felt I needed to clarify.

Memory junk can lead to unexpected behavior, bugs, or security vulnerabilities. Attackers sometimes exploit leftover sensitive data in memory if it’s not properly cleared. However, memory junk is a great tool for obfuscation but you need to be very carefuly how do you use that.

Undefined behaviour

The C programming language is a good tradeoff between readable and easily understandable code and low-level efficient code. However it contains many quirks which can be exploited. For example, this printf call functions weirdly:

```
//---CODE
#include <stdio.h>
int main() {
    printf("%s%s","mystring");
    return 0;
}
//---END CODE
```

The print function here takes in 2 parameters, each will be printed out as a string, the output will be this:

```
//---CODE
mystringN|')�
//---END CODE
```

This happens because we only supplied one argument and the second argument remains uninitialized. „N|')�” comes from the uninitialized memory that is read when printf() expects the second string argument. This data is often just leftover values from the stack or memory that the program is using, which leads to unexpected characters being printed.

Well, how can I use this for obfuscation? The answer is you can't, this is an example of undefined behaviour and first you need to understand why are these undefined behaviours are important.

Undefined behaviours can lead to segmentation faults, memory corruptions, crashes and unexpected outputs, however you can abuse this by expecting the unexpected output by carefully adding a code snippet which you tinker to your own needs to produce the good output. This helps you preserve your code functionality while adding unreadable and questionable quirks which only you understand. (Well, only for a short time)

Another code example by obfuscating with undefined behaviour:

```
//---CODE
#include <stdio.h>

int main() {
    int b; //B has no default value
    printf("%d",b); /*For test purposes,
    you don't want this in your actual code
    but this shows that the output of b is
    a memory junk like 32731.*/
    for(int i=0;i<10;i++){ //A normal for loop
        if(b!=0.1f){ /*Why is this here? Only you
know that this does nothing,
        this will always return true*/
            printf("%d,",i); //prints i like this:
0,1,2,3
        }
    }
    return 0;
}
//---END CODE
```

Classical obfuscations

The basics of the basics

First I will show you very easy to do obfuscation techinques which already introduce a lot of head scratching to reverse-engineers.

These tactics are considered bad practices by default but when it comes to obfuscation these can help a lot to make people not understand your code.

The first one is naming your variables to either one characters or naming them to bogus strings. Example for the first:

```
//---CODE

#include <stdio.h>

int main(){

    int a=0;

    int b=1;

    int c;

    for(c=0;c<5;c++){

        a+=b;

        b+=1;

    }

    printf("%d",a);

    return 0;

}

//---END CODE
```

Very nasty and hard to understand but in practice this only sums up the numbers from 1 to 5. The lack of normal naming convention and proper comments makes this code snippet very crypting and hard to understand.

The second basic method is just inconsistent tabulation or no tabulation at all. An example of the previous code with no tabulation:

```
//---CODE

#include <stdio.h>

int main(){int a=0;int b=1;int
c;for(c=0;c<5;c++){a+=b;b+=1;}printf("%d",a);return
0;}

//---END CODE
```

Now this is more unreadable than before! That means we make progress.

Some malicious actors hide code by adding a **LOT OF** spacing to code snippets making them non visible when you open and scroll in the source of the file.

Rot, atob

There are some common obfuscations and cyphers which can come handy in situations. These common cypher algorithms can come in handy when obfuscating strings or dropping and running higher level code. The first common such algorithm is the Rot (Rotation cyphering) Where you shift all the characters of a string by an amount on the ABC. The most common Rot cypher is the Rot13 as if you apply the algorithm again to a previously shifted string it will give back the original. The implemenation in C looks like this:

```
//---CODE

void rot13(char *str) {

    while (*str) {

        if ((*str >= 'A' && *str <= 'Z')) {
            *str = ((*str - 'A' + 13) % 26) + 'A';
        } else if ((*str >= 'a' && *str <= 'z')) {
            *str = ((*str - 'a' + 13) % 26) + 'a';
        }
        str++;
    }
}
//---END CODE
```

Cool, right? Here is it in practice:

```
//---CODE

#include <stdio.h>

//...Include the rot13 here...

int main() {

    char text[] = "Hello, World!";

    printf("Original: %s\n", text);

    rot13(text);

    printf("ROT13: %s\n", text);

    rot13(text);

    printf("Original again: %s\n", text);

    /*

    Output:

    Original: Hello, World!

    ROT13: Uryyb, Jbeyq!

    Original again: Hello, World!

    */

    return 0;

}

//---END CODE
```

This is not practical for hiding code in C but rather hiding strings. The rot13 is a simple algorithm for shuffling input to incomprehensible slop like in the example.

If you want to use this algorithm in practice in C/C++ you can do something like this:

```
//---CODE
/*INCLUDE rot13 from an unexpected code and rename it
to something like remove_spaces()*/
int main() {
    char text[] = "<Preivously obfuscated payload>";
    //CREATE A FILE LIKE obfuscated_script.py
    //CALL remove_spaces on the text
    //Input the text to the created file
    //Run the file.

    return 0;
}
//---END CODE
```

With this implementation you created a dropper which drops and runs a file which's content you can't see from the source code instantly. Of course if you are aware of the rot13 algorithm you may convert the slop back to the real string so this is not a big deal. Another algorithm pair is the Binary to ASCII and the ASCII to Binary (btoa and atob) pair. An example implementation of atob would look like this:

```
//---CODE
void atob(const char *str) {
    while (*str) {
        for (int i = 7; i >= 0; i--) {
            putchar(((*str >> i) & 1) ? '1' : '0');
        }
        putchar(' ');
        str++;
    }
    putchar('\n');
}

//---END CODE
```

Here is the btoa algorithm as well:

```
//---CODE

#include <ctype.h>

void btoa(const char *binStr) {

    unsigned char value = 0;

    int bitCount = 0;

    while (*binStr) {

        if (*binStr == '0' || *binStr == '1') {

            value = (value << 1) | (*binStr - '0');

            bitCount++;

            if (bitCount == 8) {

                putchar(value);

                value = 0;

                bitCount = 0;

            }

        }

        binStr++;

    }

    if (bitCount != 0) {

        printf("\nWarning: Incomplete byte
detected.\n");

    }

    putchar('\n');
}

//---END CODE
```

And in final here is the example usage of the two algorithms:

```
//---CODE

#include <stdio.h>

//Include btoa and atob here

int main() {

    char text[] = "Hello";

    printf("Binary: ");

    atob(text);

    const char *binary = "01010101 01110010 01111001
01111001 01100010";

    printf("Decoded from binary: ");

    btoa(binary);


    /*Example output: Binary: 01001000 01100101
01101100 01101100 01101111

    Decoded from binary: Uryyb*/


    return 0;

}

//---END CODE
```

Now the explanation of the following algorithm pair. The first one (atob) converts an ASCII string to binary sequences, for example the letter 'U' is 01001000 in binary. The btoa algorithm converts a binary string to an ASCII string (each byte is padded with a ' ' space letter in this example). The example code call atob to an ASCII string and btoa to a binary string as a demonstration of the algorithms.

Now you may ask the question, what did you get from knowing these. There are 2 main reasons I showed you these simple cyphers:

- Scammers and malicious actors still use atob obfuscation in Javascript, so spotting it and reverse engineering it should be easier.
- To demonstrate the easiness of making an encoder and decoder functionality.

I could've showed you base64 encoding and decoding but now that you know how encoding and decoding works you can make encoders yourself, for example a silly encoder would take in a string and reverse it. If you plug the reversed string to the same algorithm you would get back the original string. Please experiment in making your own encoder and decoder function.

Injecting dead code

Injecting dead code is still one of the most popular obfuscation and evasion tactic. It involves adding useless and never used code to your project. If you add complex and large enough amount of these dead snippets it will be way harder and tedious to reverse engineer your code. Let's use a simple code snippet for demonstration:

```
//---CODE

#include <stdio.h>

int main() {

    printf("Wow, I printed something!\n");

    return 0;

}

//---END CODE
```

Nothing special, an ordinary print inside the main function. It is obvious even for an outsider to understand this code. Now now, you can get quite creative when injecting dead code. Example:

```
//---CODE

#include <stdio.h>

int main() {

void* fgPce;

int jfjGq;

char* FoFXUp;

int a(){

    return 0;

}

printf("Wow, I printed something!\n");

return 0;

}

//---END CODE
```

Now an outside may ask these questions: “What are those variables?”, “Where does the main end?”, “What is that ‘a’ function in the middle??”. The more questions dead code injection raises the better your obfuscation is. Of course, removing the indentation can cause a bit of freakout too but you can also add gibberish comments like this:

```
//---CODE

#include <stdio.h>

int main() {

//9örj___djsk

printf("Wow, I printed something!\n");
//fwjqiu6**Ä.wjkj
return 0;
}
//---END CODE
```

This can add another layer of confusion to the previous example.

You also need to be careful with your dead code injector implementation. If all the dead code goes to the front or the back of the algorithm, experienced malware analysts can just cut out them, it is important to salt all your code equally with dead comments, variables and functions, hell, you can even make dead functions which seemingly call other dead functions. Example of a dead code injector implemented in Python:

```
#---CODE
import random
import string


def inject_dead_code(oldcode):
    lines = oldcode.split('\n')
    newcode=""
    for line in lines:
        if random.randint(0, 1)==1:
            #Add a gibberish line

line+="\n//"+''.join(random.choices(string.ascii_uppe
rcase + string.digits, k=6))+'\n'
        else:
            #Add a gibberish variable
            line+="\nvoid*
"+''.join(random.choices(string.ascii_uppercase +
string.digits, k=6))+";;\n"

        newcode+=line


    return newcode


#Tryout

code="""#include <stdio.h>

    int main() {

    printf("Wow, I printed something!");

return 0;

}"""

injected=inject_dead_code(code)

print(injected)

#---END CODE
```

This is a very basic dead code injector. Notice the double semicolons ;; this provides an extra layer of confusion. This adds unused variables and comments. An example output might look like this:

```
//---CODE

#include <stdio.h>

//3BEN3F

    int main() {

void* S2Z9NK;

    printf("Wow, I printed something!");

//15TA3X
return 0;

}

//LU0SLS

//---END CODE
```

Magnificent, isn’t it? If you want to exercise you can also implement dead function injection using the random library and extend the functionality of that simple algorithm. For further obfuscation you can also trim useful comments and remove line breaks where you can. You need to remember that removing line breaks before macros is not feasible because it will lead to macro stacking which will give you an error in most compilers.

Keep in mind that these injections might break your code so use it with caution and tweak the algorithm to fit your needs.

Control flow flattening

Control Flow Flattening (CFF) is an obfuscation technique that transforms structured control flow (e.g., if, switch, for, etc.) into a more flat and less readable structure, often by using a dispatcher loop (usually a while with a switch inside) to "flatten" the control flow. Here is the stub example where the example() function gets called inside the main:

```
//---CODE

#include <stdio.h>

void example() {

    int x = 5;

    if (x > 3) {

        printf("x is greater than 3\n");

    } else {

        printf("x is less than or equal to 3\n");

    }
    printf("End of function\n");
}
int main(){
    example();
    return 0;
}
//---END CODE
```

But when flattening the control flow, the example function turns into this mess:

```
//---CODE
#include <stdio.h>
void example() {
    int x = 5;
    int state = 0;
    while (1) {
        switch (state) {
            case 0:
                if (x > 3) {
                    state = 1;
                } else {
                    state = 2;
                }
                break;
            case 1:
                printf("x is greater than 3\n");
                state = 3;
                break;
            case 2:
                printf("x is <= 3\n");
                state = 3;
                break;
            case 3:
                printf("End of function\n");
                return;
        }
    }
}
int main(){
    example();
    return 0;
}
//---END CODE
```

The explanation is the following:

- The control flow is now "flattened" into a single while loop with a switch.
- The state variable directs the control flow instead of relying on direct structured control like if-else or block scopes.

If you want, you may expand this. Use more nesting, more switches inside larger switches, inject dead bogus states to make deobfuscation complex and introduce opaque predicates to make reverse engineering harder.

Control flow inversion

This is the last general obfuscation method and in my opinion the most useful. When used correctly this might be the most valuable tactic you can get for your buck. A very basic example for understanding:

```
//---CODE

#include <stdio.h>

int main(){

    int i=0;

    int j=0;


    while(i<10){

        j=(i+1)*2;

        printf("%d,",j);

        i++;

    }

    return 0;

}

//---END CODE
```

This code prints the numbers from 1 to 10 multiplied by 2 like this: 2,4,6,8,10...

Now without comments this may not be that obvious at first but it is still easy to figure out what this code does. With control flow inversion it will be much, much harder to tell what we are doing:

```
//---CODE

#include <stdio.h>
```

```
int main() {
    int i = 10;
    int j = 0;
    do {
        j = (11-i) * 2;
        printf("%d,", j);
        i--;
    } while (i > 0);
    return 0;
}
//---END CODE
```

Explanation: I switched the while loop to a do while loop. Common practice but make the face of the code a bit different. I also made the i iterator count downwards to induce more confusion. The output is the same but it is magnitudes harder to tell what is happening. These were all the introductory common obfuscation tactics. In the next chapter I will show you a lot of C language specific obfuscation tactics which may come handy once.

Obfuscation in C

The GOTO in the room

C supports the goto instruction. The goto instruction is very divisive among the software developers as it is not considered structured. I am fine with it and sometimes I use it because it make more sense than making a loop. What does the goto instruction do? It jumps to a defined label. Labels can be defined like this:

```
//---CODE
```

```
#include <stdio.h>

int main() {

    my_label:

//Code there

    return 0;

}

//---END CODE
```

Labels can act as code structuring as they can be used to name segments of the code. You can jump to a label using a goto instruction like this:

```
//---CODE

    goto label;

//---END CODE
```

This is practicularly useful in cases where you want to invert the control flow as nobody wants to debug 10 or 100 nested goto calls.

You can do some quirky things using the goto instruction. For starting here is a function which uses a for loop to print the numbers from 0-9:

```
//---CODE

#include <stdio.h>

int main() {

    for(int i=0;i<10;i++){

        printf("%d",i);

    }

    /*Output: 0123456789*/

    return 0;

}
```

```
//---END CODE
```

That for loop can be substituted to a label and a goto call like this:

```
//---CODE
#include <stdio.h>
int main() {
    int i=0;
    forlabel:
        printf("%d",i);
    if(i+1<10){
        ++i;
        goto forlabel;
    }
    /*Output: 0123456789*/
    return 0;
}
//---END CODE
```

This program simulates a for loop with a goto, the i counter is declared and if it is lesser than the desired amount we increment it and perform a goto jump. You can simulate for loops with goto easily. It turns out, you can simulate **any** loop with goto easily.

What are goto calls capable of:

- Jumping from a for loop to inside another for loop. Causing the second for loop counter to be undefined.
- Perform jumps to one after another and make the code incredibly cryptic.
- Annoy people who want to reverse engineer it.

Allow me to prove my second point with an obfuscated hello world example:

```
//---CODE

#include <stdio.h>

int main() {

pAsMQqKRjtGHJB:

goto WdfZqhPAvKydlV;WdfZqhPAvKydlV:

goto LJFYZMJmejchLu;LJFYZMJmejchLu:

goto TPFeKDsdEkzHDk;TPFeKDsdEkzHDk:

goto hudwColGKiHXyg;hudwColGKiHXyg:

goto DQSkhOqIpbPNpO;DQSkhOqIpbPNpO:

goto CdIAUopQTJcJjf;CdIAUopQTJcJjf:

goto OHIXrIXvRGkElG;OHIXrIXvRGkElG:

goto cDFxkfyRgZbGvi;cDFxkfyRgZbGvi:

goto WCKlCNfkcRpwvf;WCKlCNfkcRpwvf:

goto KRRsxOBWIDFcxr;KRRsxOBWIDFcxr:

goto MsMENtSAIoJdxWO;MsMENtSAIoJdxWO:

printf("Hello world");

WCKlCufkcRpwvf:

return 0;

}

//---END CODE
```

Conbine this with the dead code injector and you got yourself a baddie. This is an example, easier application but the complexity can vary between obfuscator implementations. Use it wisely!

Obfuscation by obscurity

This part will cover the obscure part of C. This will not only help you understand these keywords but will also give you the knowledge to distinguish bogus code from real useful code. For example, if you need to reverse-engineer a code you will instantly be able to realize what keywords are here just for making the code harder to read. For example in most cases you can just CTRL+H all the occurences of these outdated keywords and remove them because nothing will happen if you remove them other than the code being more clean and more reverse-engineerable.

Obfuscation by obscurity is a strategy where by using lesser known features of a language you obscure your code and thus making it harder for outsiders to understand your code. This sounds great but in practice it does very little to protect your code. It may induce a "Huh?" moment in the person that is trying to understand your code but it only costs them a single google search to figure out the bogus. In a later chapter we will build on these tactics but we will also learn more obscure tactics that can't be googled easily.

Our first patient is the "register" keyword. In older times this keyword suggested for the compiler that the variable should be stored in a CPU register for faster access. Nowdays most compilers ignore it because most of them is already quite efficient in optimizing code and deciding what variables should be put in a register and thus, the register keyword became obscure. But obsolence does not mean you can't strap it to your variables and make the program function exactly the same:

```
//---CODE

#include <stdio.h>

int main() {

    register int x=1;

    for(register int i=0;i<10;i++){

        printf("%d,",x);

        x*=2;
    }
    return 0;
}
//---END CODE
```

This program outputs the powers of two,however it is much more painful to look at the code because of the register keywords. It is important to note that the register keyword HAD some effects in older times but now it gets completely ignored so nothing is stopping you from stuffing your code with register keywords all the way.

The second confusing patient is the restrict keyword which can be used for function arguments. It tells the compiler that for the lifetime of the pointer its value shall be accessed from that pointer or a data derived from it.

```
//---CODE
#include <stdio.h>
void add(int* restrict a, int b){
    *a+=b;
}
int main() {
    int x=1;
    add(&x,2);
    printf("%d",x);
    return 0;
}
//---END CODE
```

This is our revised add function which uses the restrict keyword but it is useless here, only here for artistic purposes. But for example if you have a string copy which takes in a source and destination string pointer, having the restrict keywords there can suggest the compiler that these 2 pointers do not overlap so it can apply more aggressive optimization.

The _Noreturn keyword was introduced in C11 and indicates that the function will not return any value, an example:

```
//---CODE

#include <stdio.h>

#include <stdlib.h>

#include <stdnoreturn.h>

_Noreturn void add(int* a, int b){

    *a+=b;

    exit(1);

}

int main() {

    int x=1;

    add(&x,2);

    printf("%d",x);

    return 0;

}

//---END CODE
```

This sounds very impractical because IT IS but adding it to your main function and exitting with an exit call can be a very clever way to make your code more obscure. For practice make a code snippet which heavily uses all of these and look at the final result with your own eyes.

Lesser known but useful keywords

In this section I will tell you about a bunch of lesser known but useful keywords which you can use in your further projects. This is not necessarily obfuscation but remember: There is nothing stopping you from strapping these keywords to everywhere you please. Also knowing about these will help you distinguish between geniune optimizations and obfuscation methods.

The first keyword is the volatile keyword which in my opinion one of the most useful keywords when working with low level C. It tells the compiler to not optimize a code segment which otherwise would be optimized or be removed. (Because the compiler finds it redundant). An example usage:

```
//---CODE

int main() {

    volatile int reg1=0; //Not used anywhere

    volatile int reg2=1;

    if(reg2==0){ /*Never runs but also does not get
removed because of the volatile keyword*/

        return 1;

    }

    return 0;

}

//---END CODE
```

By default the compiler would know that the reg1 and reg2 have no practical meanings here because reg2 has a redundant value and declaring reg1 is also redundant. However it does not remove them and still compiles them into the destination file because we specifically told it to not remove those using the volatile keyword.

The second keyword is the _Atomic keyword which was instrocuded in C11. This keyword makes a variable as atomic, which provides atomic operations for concurrency without using locks. This guarantees that operations are on the marked variable are atomic, meaning that no thread sees it as an inconsistent state.

```
//---CODE

#include <stdatomic.h>

#include <stdio.h>

int main() {

    _Atomic int x=1; //Works as normal

    printf("%d",x);

    return 0;

}

//---END CODE
```

Multi-threading is not something this book covers but for now all you need to do that this keyword is a valid keyword. The last 2 sections are not only useful for obfuscation but also useful for knowing just a little more about the C language.

```
//---CODE

_Atomic volatile register int x=1;

//---END CODE
```

If you encounter this then you will know exactly what are those keywords for. In the later chapter, I will show you how to make keywords-like bogus keywords which look like this but no amount of google search will tell you about them (Because they are bogus).

The __attribute__ keyword

The __attribute__ keyword is well, used to tell the attributes about the next variable or function or method to the compiler. It is more useful in low-level programming. Here are some examples which are actually useful:

```
//---CODE

__attribute__((noreturn)) /*Indicates that the next
function has no return value*/

__attribute__((packed)) /*Indicates that the next
struct or variable shall not be optimized and
redundant values should be kept.*/

__attribute__((optimize("Ofast,unroll-loops")))
/*Indicates the optimization level and options the
next function should have*/

__attribute__((always_inline)) /*Indicates the the
next function or variable should be always threated
as an inline member.*/

//---END CODE
```

Understanding these is not necessary because each compiler has their own attribute macro like this. What is more important is you can use some bogus value inside these attributes. Here is an example:

```
//---CODE

#include <stdio.h>

__attribute__((packed)) int x=0;

int main(){

    /*Output: warning: 'packed' attribute ignored [-
Wattributes]

    3 | __attribute__((packed)) int x=0;*/

    return 0;
```

```c
}

//---END CODE
```

Attributes can only be defined outside functions. If it does not have any inherent meaning using it will give a warning but not an error, so it is fine. The first usage of this keyword in obfuscation is just stuffing a lot of attributes to a variable like this:

```c
//---CODE

#include <stdio.h>

__attribute__((visibility("default")))
__attribute__(()) __attribute__((format(printf, 1,
2))) __attribute__((malloc, alloc_size(1)))
__attribute__((cold)) __attribute__((hot))
__attribute__((optimize("Ofast,unroll-loops")))
__attribute__((no_reorder))
__attribute__((externally_visible))
__attribute__((no_sanitize_address))
__attribute__((warn_unused_result))
__attribute__((pure)) __attribute__((noinline))
__attribute__((always_inline)) __attribute__((const))
__attribute__((naked)) __attribute__((fastcall))
__attribute__((constructor))
__attribute__((destructor)) __attribute__((weak))
__attribute__((noreturn)) __attribute__((packed,
aligned(128))) __attribute__((section(".text"))) int
x=0;

int main(){

    //Gives a lot of warnings but compiles

    printf("%d",x); //Still works

    return 0;

}

//---END CODE
```

Note that this may break your code and should be only applied to your global variables. This already looks bad but if you have more variables this will turn worse. and if you implement junk variables and functions using the dead code injector this becomes hell.

What if I told you that this can be even worse? The attribute keyword does not care what you put inside it. So this is an fully valid code with 1 warning:

```
//---CODE

#include <stdio.h>

__attribute__((this_bogus_does_not_exist)) int x=0;

int main(){

    printf("%d",x); //Still works

    return 0;

}

//---END CODE
```

With this newfound knowledge mixed with the dead code injection you can truly make hell for anyone who wants to understand or reverse-engineer your code.

It is also important to note that having a attributes stacked with the same value may lead to the program not compiling in some compilers like gcc. For example:

```
//---CODE

__attribute__((visibility("default")))
__attribute__((visibility("default"))) int x=5;
//This never compiles

//---END CODE
```

Avoid this. If you want to practice you can make a python function which appends a lot of random attributes to function and variables located in the global scope.

Macros

The C programming language offers a preprocessor which can do some things like including other source or header files, define a macro and much more. If you programmed in C before you have probably encountered the #include macro which is used to include a source file, however this is just one useful macro among the several preprocessor directives.

The first macro that this guide will talk about is the #define macro which is probably the single best tool if you want to obfuscate your C program. It is important to note that anybody can run the preprocessor on your C codeto reveal the code hidden behind the obfuscated macros.

The define macro is used to make another label which references a value. An example of defining PI:

```
//---CODE

#include <stdio.h>

#define PI 3.1415926535

int main(){

    printf("Value of PI: %lf",PI);

    return 0;

}

//---END CODE
```

This can be very useful for avoiding magic numbers but you can define a constant double globally if you don't want to use the #define keyword.

Another use of the define keyword is to make aliases to different keywords. Example:

```
//---CODE

#include <stdio.h>

#define my_int int
```

```
#define my_if if
my_int main(){
    my_int i=1;
    my_if(i==1){
        printf("%d",i);
    }
    return 0;
}
//---END CODE
```

The preprocessor notices the definitions and replaces it with the corresponding value. For example my_int will become int and my_if will become if. You can define almost ANY keyword and value. For excercise you can make a python function which encodes common keywords to gibberish. An example output would be this:

```
//---CODE
#define kFPwZ if
#define XSgay while
#define GcXIM do
#define YNYwi for
#define tDcyz void
#define zayrS int
#define TgJbm char
#define GknpF switch
#define SOycv return
//---END CODE
```

This works because the preprocessor inserts the defined correct values but it will be insanely hard to read this garbage code before processing.

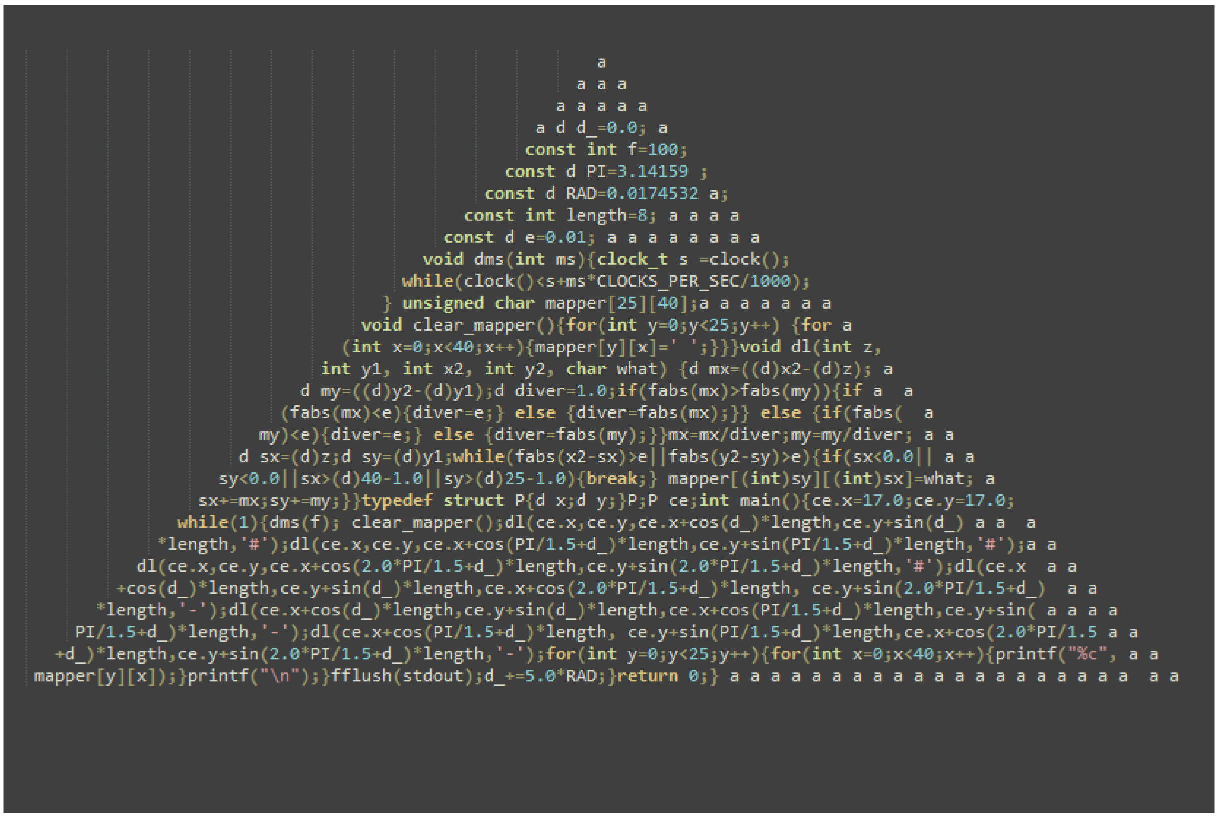
There are several other obfuscation methods which involves using the define directive. For example you can define a random bogus keyword with no value. It will be defined but threated as an empty keyword. For example you can do this:

```
//---CODE

#define a

//---END CODE
```

You may wonder why is this useful? Well, for making art like this it is pretty much necessary:



```
                                                       a
                                                     a a a
                                                   a a a a a
                                                 a d d_=0.0; a
                                                const int f=100;
                                              const d PI=3.14159 ;
                                            const d RAD=0.0174532 a;
                                          const int length=8; a a a a
                                        const d e=0.01; a a a a a a a a
                                      void dms(int ms){clock_t s =clock();
                                    while(clock()<s+ms*CLOCKS_PER_SEC/1000);
                                  } unsigned char mapper[25][40];a a a a a a a
                                void clear_mapper(){for(int y=0;y<25;y++) {for a
                              (int x=0;x<40;x++){mapper[y][x]=' ';}}}void dl(int z,
                            int y1, int x2, int y2, char what) {d mx=((d)x2-(d)z); a
                          d my=((d)y2-(d)y1);d diver=1.0;if(fabs(mx)>fabs(my)){if a  a
                        (fabs(mx)<e){diver=e;} else {diver=fabs(mx);}} else {if(fabs(  a
                      my)<e){diver=e;} else {diver=fabs(my);}}mx=mx/diver;my=my/diver; a a
                    d sx=(d)z;d sy=(d)y1;while(fabs(x2-sx)>e||fabs(y2-sy)>e){if(sx<0.0|| a a
                  sy<0.0||sx>(d)40-1.0||sy>(d)25-1.0){break;} mapper[(int)sy][(int)sx]=what; a
                sx+=mx;sy+=my;}}typedef struct P{d x;d y;}P;P ce;int main(){ce.x=17.0;ce.y=17.0;
              while(1){dms(f); clear_mapper();dl(ce.x,ce.y,ce.x+cos(d_)*length,ce.y+sin(d_) a a  a
            *length,'#');dl(ce.x,ce.y,ce.x+cos(PI/1.5+d_)*length,ce.y+sin(PI/1.5+d_)*length,'#');a a
          dl(ce.x,ce.y,ce.x+cos(2.0*PI/1.5+d_)*length,ce.y+sin(2.0*PI/1.5+d_)*length,'#');dl(ce.x  a a
        +cos(d_)*length,ce.y+sin(d_)*length,ce.x+cos(2.0*PI/1.5+d_)*length, ce.y+sin(2.0*PI/1.5+d_)  a a
      *length,'-');dl(ce.x+cos(d_)*length,ce.y+sin(d_)*length,ce.x+cos(PI/1.5+d_)*length,ce.y+sin( a a a a
    PI/1.5+d_)*length,'-');dl(ce.x+cos(PI/1.5+d_)*length, ce.y+sin(PI/1.5+d_)*length,ce.x+cos(2.0*PI/1.5 a a
  +d_)*length,ce.y+sin(2.0*PI/1.5+d_)*length,'-');for(int y=0;y<25;y++){for(int x=0;x<40;x++){printf("%c", a a
mapper[y][x]);}printf("\n");}fflush(stdout);d_+=5.0*RAD;}return 0;} a a a a a a a a a a a a a a a a a a a a  a a
```

Pretty, isn't it? This snippet of code makes a 3d spinning pyramid inside your console when ran. This compiles and runs perfectly because the we defined the „a" symbol to nothing („") using the define directive.

But wait, there is more! Remember the _Atomic and _Noreturn values? We can make our keyword lookalikes using the define directive:

```
//---CODE

#define _Bitonic

#define Unsafe

#define _Parallel

#define _Invalid

//---END CODE
```

This is completely bogus but sounds very C like, no amount of google searches will show others the meaning of your new keywords. The full code would look something like this:

```
//---CODE

#include <stdio.h>

#define _Bitonic

#define Unsafe

#define _Parallel

#define _Invalid


int main() {

    _Bitonic Unsafe _Parallel register int x=5;

    _Invalid printf("Value of x: %d\n",x);

    /*Expected output: Value of x: 5*/

    return 0;

}

//---END CODE
```

Just remember to include your „custom” keywords from a random header to make debugging harder and more time consuming. You can also add a functionality to your python obfuscator script that adds #define <random string> to your code and just places these bogus keywords all over the place. It wont harm your code since it is defined as nothing but make sure to add a space before and after the inserted bogus keyword.

The next useful obfuscation method is using trigraphs instead of the normal # macro signal. For example you can make this code:

```
//---CODE

#include <stdio.h>

//---END CODE
```

Look like this:

```
//---CODE

??=include <stdio.h>

//---END CODE
```

The ??= symbol is called a trigraph and can be used as a substitute for the # symbol. You may need to add additional flags when compiling your code.

If you want to reverse-engineer literally anything from this section just run this command:

```
gcc -E source.c
```

This will apply the preprocessor to your file allowing you to see through the bogus easily.

Obfuscation in C++

Lambda functions

Lambda functions are a great addition to C++. They are most useful when you are dealing with callbacks, however this guide does not focus on the usefulness. As you could've guessed we are going to be focusing on how cryptic you can make your code with lambda functions.

The layout of a basic lambda function looks like this:

```
//---CODE

[/*How should the external variables be threated, & means reference, that is the only option you need to know for now.*/](/*Parameters*/){/*Logic*/}

//---END CODE
```

This is an example use of a lambda function where the lambda function doubles the given argument:

```
//---CODE

#include <iostream>

int main() {

    auto double_it = [](int x) {

        return x *2;

    };

    std::cout << double_it(5);

    return 0;

}

//---END CODE
```

This is all good, but how can we use these to hide our logic? The answer is very simple, we wrap every possible code snippet into more lambda function, inject empty lambda function or even inject dead, never called, useless lambda functions:

```
//---CODE
#include <iostream>
int main() {
    [](float bnwsj) {
        return bnwsj *57823748;
    };

    [](){}; //Add useless lambda function
    auto call=[](){
        auto double_it = [](int x) {
            return x *2;
        };
        std::cout << double_it(5);


    };
    call();
    return 0;
}
//---END CODE
```

Quite simple, combining this with the other obfuscation methods already produces a big mess but the next chapter will allow us to make more out of these lambda functions.

The [[]] madness

The [[]] macro is simillar tot he __attribute__ keyword but not all C compilers support it. It is more like a C++ feature than a C feature. They have simillar purpose and use cases. Example use of [[]]:

```
//---CODE

[[gnu::always_inline]] …

[[maybe_unused]] int unused_var=5;

//---END CODE
```

While these are good and valid use cases (commonly used for optimization) you can use them for more sinister purposes. Lets start with this code snippet and then descend slowly into hell:

```
//---CODE

#include <iostream>

int p=5;

int main() {

    std::cout << "Value of p: " << p;

    return 0;

}

//---END CODE
```

This code just declares a global variable p, and in the main the variable gets printed out. It turns out that you can just append [[]] before any function and variable declaration and there will be no warnings (at least using g++ for compilation). It also turns out that you can stack unlimited amount of [[]]-s before like this:

```
//---CODE

#include <iostream>
```

```
[[]] [[]] [[]] [[]] [[]] [[]] [[]] int p=5;
[[]] [[]] [[]] [[]] [[]] [[]] [[]] int main() {
    std::cout << "Value of p: " << p;
    return 0;
}
//---END CODE
```

This still works. But I think you see my point. You can also add any string tot he inside of the [[]] macro like this:

```
//---CODE
#include <iostream>
[[zuzuzg8]] [[hgjgj]] int p=5;
[[]] [[gjgjgj]] int main() {
    std::cout << "Value of p: " << p;
    return 0;
}
//---END CODE
```

While this does produce warnings (because the compiler ignores these bogus directives), warnings are not errors after all so we are fine. What if I told you that you can also abuse the and() reserved keyword to nest lambda functions and inner [[]]-s to your code? This looks hilarious:

```
//---CODE
#include <iostream>
[[and([](){}[[({&&})]]])]] [[and([](){ ;fwWqr 42;
}[[({&&, ||, ~, []{}()})]]])]] int p=5;
int main() {
    std::cout << "Value of p: " << p;
    return 0;

}

//---END CODE
```

The warnings still persists because the compiler does not like what we are doing here but still no errors. With this said this code is valid because we pass a lambda function to the and() call. The 2 example lambda functions (yes, there are 2 of them) Are valid lambda functions by themselves. For the ending here is a small „motivation” of what happens when we apply most of the learnt tactics to printing out a global variable:

```
//---CODE

??=define PjjUW if

??=define TjNdT while

??=define UxfiB do

??=define cYujt for

??=define wOCbP void

??=define ScotY int

??=define DunIq char

??=define xIxlr switch

??=define yUlrY return

??=include <stdio.h>

[[]] [[]] [[]] [[]] [[]] [[and([](){}[[({&&})]])]]
[[and([](){ ;yUlrY 42; }[[({&&, ||, ~, []{}()})]])]]
[[gnu::deprecated("AAA"), gnu::warning("AAA"),
gnu::used,  gnu::assume_aligned(64),
gnu::no_reorder,gnu::unused,gnu::pure, gnu::flatten,
gnu::cold,gnu::packed, gnu::malloc, gnu::artificial,
gnu::aligned(128), gnu::noinline, gnu::noclone]]
[[gnu::always_inline]] [[gnu::hot]] [[gnu::const]]
[[using CC: opt(1), debug, optimize(3),
visibility("default")]]
```

```
[[maybe_unused,likely,noreturn,carries_dependency,dep
recated("AAA"),fallthrough,nodiscard("MMMM"),
unlikely,no_unique_address,optimized,assume(1),indete
rminate,optimized_for_synchronized]]
[[msvc::forceinline, clang::optnone, noduplicate,
returns_nonnull, restrict, trivial_abi,
translatable]] __attribute__((visibility("default")))
__attribute__(()) __attribute__((format(printf, 1,
2))) __attribute__((malloc, alloc_size(1)))
__attribute__((cold)) __attribute__((hot))
__attribute__((optimize("Ofast,unroll-loops")))
__attribute__((no_reorder))
__attribute__((externally_visible))
__attribute__((no_sanitize_address))
__attribute__((warn_unused_result))
__attribute__((pure)) __attribute__((noinline))
__attribute__((always_inline)) __attribute__((const))
__attribute__((naked)) __attribute__((fastcall))
__attribute__((constructor))
__attribute__((destructor)) __attribute__((weak))
__attribute__((noreturn)) __attribute__((packed,
aligned(128))) __attribute__((section(".text")))
[[Xdxic::ADKOl]] alignas(128) extern inline volatile
;ScotY p=5;

[[]] [[]] [[]] [[]] [[]] [[and([](){}[[({&&})]]])]]
[[and([](){ ;yUlrY 42; }[[({&&, ||, ~, []{}()})]]])]]
[[gnu::deprecated("AAA"), gnu::warning("AAA"),
gnu::used,  gnu::assume_aligned(64),
gnu::no_reorder,gnu::unused,gnu::pure, gnu::flatten,
gnu::cold,gnu::packed, gnu::malloc, gnu::artificial,
gnu::aligned(128), gnu::noinline, gnu::noclone]]
[[gnu::always_inline]] [[gnu::hot]] [[gnu::const]]
[[using CC: opt(1), debug, optimize(3),
visibility("default")]]
[[maybe_unused,likely,noreturn,carries_dependency,dep
recated("AAA"),fallthrough,nodiscard("MMMM"),
unlikely,no_unique_address,optimized,assume(1),indete
rminate,optimized_for_synchronized]]
[[msvc::forceinline, clang::optnone, noduplicate,
returns_nonnull, restrict, trivial_abi,
translatable]] __attribute__((visibility("default")))
__attribute__(()) __attribute__((format(printf, 1,
2))) __attribute__((malloc, alloc_size(1)))
__attribute__((cold)) __attribute__((hot))
__attribute__((optimize("Ofast,unroll-loops")))
__attribute__((no_reorder))
__attribute__((externally_visible))
__attribute__((no_sanitize_address))
__attribute__((warn_unused_result))
__attribute__((pure)) __attribute__((noinline))
__attribute__((always_inline)) __attribute__((const))
__attribute__((naked)) __attribute__((fastcall))
__attribute__((constructor))
__attribute__((destructor)) __attribute__((weak))
__attribute__((noreturn)) __attribute__((packed,
aligned(128))) __attribute__((section(".text")))
__attribute__((UevAA)) alignas(128) register inline
volatile  ;ScotY main(){…

//---END CODE
```

This is not the whole code but at the end we have achieved quite a good run! Pat yourself on the back and congratulations for getting this far!

Epilogue

Thank you for reading my guide. This really only covered the basics of obfuscation but I hope you learnt a lot about C and C++ and their obscure behaviours. The was really only the basics, if you want to learn more see my online obfuscator website:

https://afghangoat.hu/obfuscator

With that out of the way thank you for reading my guide again and have a nice day!

Credits

Credits for my university mate, Tamás Csordás, who helped me figure out and imporve the [[]] obfuscation method drastically.

What is next?

If you still feel like you need to practice try to expand you obfuscator to perhaps even more languages. Learning Javascript and Python obfuscation and deobfuscation is also a great addition because you will be able to reverse-engineer most online scams and spams.