Confusing Your Tutor

Introduction to C Obfuscation

Nicholas Berridge-Argent

2021

"Don't do this, don't make your tutor cry."
— Dr. Andrew Taylor, COMP1511 18s1

Contents

What is Obfuscation / Golf?

What is Obfuscation / Golf?

- *obfuscation* (n.) — the action of making something obscure, unclear, or unintelligible. (Oxford Languages)

What is Obfuscation / Golf?

- *obfuscation* (n.) — the action of making something obscure, unclear, or unintelligible. (Oxford Languages)
- In short, making code harder to read.

What is Obfuscation / Golf?

- *obfuscation* (n.) — the action of making something obscure, unclear, or unintelligible. (Oxford Languages)
- In short, making code harder to read.
- A slightly separate concern: code golf.

What is Obfuscation / Golf?

- *obfuscation* (n.) — the action of making something obscure, unclear, or unintelligible. (Oxford Languages)
- In short, making code harder to read.
- A slightly separate concern: code golf.
- Writing code using as few characters as possible.

Why Obfuscate?

Why Obfuscate?

- Distribute your source code but keep it secret (JavaScript).

Why Obfuscate?

- Distribute your source code but keep it secret (JavaScript).
- For fun (IOCCC) / to annoy people.

Why Obfuscate?

- Distribute your source code but keep it secret (JavaScript).
- For fun (IOCCC) / to annoy people.
- To learn the limits of what a language can do.

Why Golf?

Why Golf?

- Speed up sending code over the internet and save bandwidth (Webpack).

Why Golf?

- Speed up sending code over the internet and save bandwidth (Webpack).
- Type scripts quickly on the command line (Perl).

Why Golf?

- Speed up sending code over the internet and save bandwidth (Webpack).
- Type scripts quickly on the command line (Perl).
- For fun (code golf contests) / to annoy people.

Why Golf?

- Speed up sending code over the internet and save bandwidth (Webpack).
- Type scripts quickly on the command line (Perl).
- For fun (code golf contests) / to annoy people.
- To learn the limits of what a language can do.

Why Not?

Why Not?

- There are reasons you may not want to obfuscate (or golf code).

Why Not?

- There are reasons you may not want to obfuscate (or golf code).
- With regards to university work:

Why Not?

- There are reasons you may not want to obfuscate (or golf code).
- With regards to university work:
  - In an assignment, you lose style marks.

Why Not?

- There are reasons you may not want to obfuscate (or golf code).
- With regards to university work:
  - In an assignment, you lose style marks.
  - In an exam, your marker gives up on reading your code.

Why Not?

- There are reasons you may not want to obfuscate (or golf code).
- With regards to university work:
  - In an assignment, you lose style marks.
  - In an exam, your marker gives up on reading your code.
- You lose maintainability, portability, debug-ability. You risk introducing errors.

Why Not?

- There are reasons you may not want to obfuscate (or golf code).
- With regards to university work:
  - In an assignment, you lose style marks.
  - In an exam, your marker gives up on reading your code.
- You lose maintainability, portability, debug-ability. You risk introducing errors.
- For C in particular, most techniques are defeated by the compiler...

Why Not?

Why Not?

Let's look at a normal and obfuscated Fibonacci sequence generator.

Why Not?

Let's look at a normal and obfuscated Fibonacci sequence generator.

```
int fib(int n) {
  int a = 0;
  int b = 1;

  for (int i = 0; i < n - 1; i++) {
    int tmp = a + b;
    a = b;
    b = tmp;
  }

  return b;
}
```

Why Not?

Let's look at a normal and obfuscated Fibonacci sequence generator.

```c
int fib(int n) {
  int a = 0;
  int b = 1;

  for (int i = 0; i < n - 1; i++) {
    int tmp = a + b;
    a = b;
    b = tmp;
  }

  return b;
}
```

```c
int fib(int n) {
  int b = 1;

  for (int a = b - 1, i = a;
    i++ < n - 1;
    b = a + b, a = b - a);

  return b;
}
```

Why Not?

Let's look at a normal and obfuscated Fibonacci sequence generator.

```c
int fib(int n) {
  int a = 0;
  int b = 1;

  for (int i = 0; i < n - 1; i++) {
    int tmp = a + b;
    a = b;
    b = tmp;
  }

  return b;
}
```

```c
int fib(int n) {
  int b = 1;

  for (int a = b - 1, i = a;
    i++ < n - 1;
    b = a + b, a = b - a);

  return b;
}
```

Then we'll compile them with optimisation enabled (`-O`) and inspect the assembly code (`-S`):

Why Not?

Let's look at a normal and obfuscated Fibonacci sequence generator.

```
int fib(int n) {
  int a = 0;
  int b = 1;

  for (int i = 0; i < n - 1; i++) {
    int tmp = a + b;
    a = b;
    b = tmp;
  }

  return b;
}
```

```
int fib(int n) {
  int b = 1;

  for (int a = b - 1, i = a;
    i++ < n - 1;
    b = a + b, a = b - a);

  return b;
}
```

Then we'll compile them with optimisation enabled (`-O`) and inspect the assembly code (`-S`):

```
clang -S -O -o fibonacci.s fibonacci.c
clang -S -O -o fibonacci-obf.s fibonacci-obf.c
```

Why Not?

Why Not?

```
        .text
        .file   "fibonacci.c"
        .globl  fib
        .p2align        4, 0x90
        .type   fib,@function
fib:
        .cfi_startproc
# %bb.0:
        movl    $1, %eax
        cmpl    $2, %edi
        jl      .LBB0_3
# %bb.1:
        addl    $-1, %edi
        movl    $1, %ecx
        xorl    %edx, %edx
        .p2align        4, 0x90
.LBB0_2:
        movl    %edx, %eax
        addl    %ecx, %eax
        movl    %ecx, %edx
        movl    %eax, %ecx
        addl    $-1, %edi
        jne     .LBB0_2
.LBB0_3:
        retq
```

```
        .text
        .file   "fibonacci-obf.c"
        .globl  fib
        .p2align        4, 0x90
        .type   fib,@function
fib:
        .cfi_startproc
# %bb.0:
        movl    $1, %eax
        cmpl    $2, %edi
        jl      .LBB0_3
# %bb.1:
        addl    $-1, %edi
        movl    $1, %ecx
        xorl    %edx, %edx
        .p2align        4, 0x90
.LBB0_2:
        movl    %edx, %eax
        addl    %ecx, %eax
        movl    %ecx, %edx
        movl    %eax, %ecx
        addl    $-1, %edi
        jne     .LBB0_2
.LBB0_3:
        retq
```

Why Not?

Why Not?

- The resulting assembly code is exactly the same after optimisation!

Why Not?

- The resulting assembly code is exactly the same after optimisation!
- The compiler is a lot smarter than we are.

Why Not?

- The resulting assembly code is exactly the same after optimisation!
- The compiler is a lot smarter than we are.
- So this can't actually stop people from figuring out what the *compiled* program is doing.

Why Not?

- The resulting assembly code is exactly the same after optimisation!
- The compiler is a lot smarter than we are.
- So this can't actually stop people from figuring out what the *compiled* program is doing.
- There are other ways to do that, which this workshop is too narrow to contain.

Goals

Goals

This workshop has several goals:

Goals

This workshop has several goals:

- Teach you some things you (probably) didn't know about C.

Goals

This workshop has several goals:

- Teach you some things you (probably) didn't know about C.
- Give you some tools and techniques you can use to obfuscate or golf your own code.

Goals

This workshop has several goals:

- Teach you some things you (probably) didn't know about C.
- Give you some tools and techniques you can use to obfuscate or golf your own code.
- Get you to try out these tools and techniques.

Goals

This workshop has several goals:

- Teach you some things you (probably) didn't know about C.
- Give you some tools and techniques you can use to obfuscate or golf your own code.
- Get you to try out these tools and techniques.
- Show you some worked, finished examples.

Goals

This workshop has several goals:

- Teach you some things you (probably) didn't know about C.
- Give you some tools and techniques you can use to obfuscate or golf your own code.
- Get you to try out these tools and techniques.
- Show you some worked, finished examples.

I'm not an expert in C so I can't show you everything, but I should be able to at least show you some things.

Prerequisites

Prerequisites

- All the content is designed to be accessible to fresh COMP1511 graduates.

Prerequisites

- All the content is designed to be accessible to fresh COMP1511 graduates.
- We will cover a few examples relating to MATH11[34]1, MATH1081, COMP1521, but you don't need to have taken these courses.

Prerequisites

- All the content is designed to be accessible to fresh COMP1511 graduates.
- We will cover a few examples relating to MATH11[34]1, MATH1081, COMP1521, but you don't need to have taken these courses.
- If you want to follow along, you should have an editor / IDE and a C compiler at the ready.

Intended Schedule

Intended Schedule

This workshop is designed to go overtime by one hour (oops...) but you should get the most out of the first two hours:

Intended Schedule

This workshop is designed to go overtime by one hour (oops...) but you should get the most out of the first two hours:

1. First hour: Obfuscation techniques.

Intended Schedule

This workshop is designed to go overtime by one hour (oops...) but you should get the most out of the first two hours:

1. First hour: Obfuscation techniques.
2. Second hour: Writing programs to obfuscate.

Intended Schedule

This workshop is designed to go overtime by one hour (oops...) but you should get the most out of the first two hours:

1. First hour: Obfuscation techniques.
2. Second hour: Writing programs to obfuscate.
3. Third hour: Worked examples.

Intended Schedule

This workshop is designed to go overtime by one hour (oops...) but you should get the most out of the first two hours:

1. First hour: Obfuscation techniques.
2. Second hour: Writing programs to obfuscate.
3. Third hour: Worked examples.

Between each of these we'll take a quick 5 minute break.

Contents

Comments, Identifiers and Constants

Comments, Identifiers and Constants

The kinds of things you would normally lose style marks on by mistake. We can remove comments and `#define`'d constants, and make all identifiers 1-2 letters long.

Comments, Identifiers and Constants

The kinds of things you would normally lose style marks on by mistake. We can remove comments and `#define`'d constants, and make all identifiers 1-2 letters long.

```
// Simple to read

#define FACTOR 42
int eliminate_divisible(int * numbers,
                        int size)
{
  int count = 0;

  for (int i = 0; i < size; i++) {
    if (numbers[i] % FACTOR == 0) {
      numbers[i] = 0;

      // Record how many numbers we
      // eliminate
      count++;
    }
  }

  return count;
}
```

Comments, Identifiers and Constants

The kinds of things you would normally lose style marks on by mistake. We can remove comments and `#define`'d constants, and make all identifiers 1-2 letters long.

```c
// Simple to read

#define FACTOR 42
int eliminate_divisible(int * numbers,
                        int size)
{
  int count = 0;

  for (int i = 0; i < size; i++) {
    if (numbers[i] % FACTOR == 0) {
      numbers[i] = 0;

      // Record how many numbers we
      // eliminate
      count++;
    }
  }

  return count;
}
```

```c
// Is it still obvious what this is doing?

int e(int * n, int s)
{
  int c = 0;

  for (int i = 0; i < s; i++) {
    if (n[i] % 42 == 0) {
      n[i] = 0;
      c++;
    }
  }

  return c;
}
```

Obfuscation Quiz 1: MATH1081

Obfuscation Quiz 1: MATH1081

What does this function return if called as `e(47, 19)`?

```
int e(int a, int b)
{
  while (b) {
    int q = b;
    b = a % b;
    a = q;
  }

  return a;
}
```

Obfuscation Quiz 1: MATH1081

What does this function return if called as `e(47, 19)`?

```
int e(int a, int b)
{
  while (b) {
    int q = b;
    b = a % b;
    a = q;
  }

  return a;
}
```

1. 1

Obfuscation Quiz 1: MATH1081

What does this function return if called as `e(47, 19)`?

```
int e(int a, int b)
{
  while (b) {
    int q = b;
    b = a % b;
    a = q;
  }

  return a;
}
```

1. 1
2. 7

Obfuscation Quiz 1: MATH1081

What does this function return if called as `e(47, 19)`?

```
int e(int a, int b)
{
  while (b) {
    int q = b;
    b = a % b;
    a = q;
  }

  return a;
}
```

1. 1
2. 7
3. Infinite loop

Obfuscation Quiz 1: MATH1081

What does this function return if called as `e(47, 19)`?

```
int e(int a, int b)
{
  while (b) {
    int q = b;
    b = a % b;
    a = q;
  }

  return a;
}
```

1. 1
2. 7
3. Infinite loop
4. 0

Obfuscation Quiz 1: MATH1081

What does this function return if called as `e(47, 19)`?

```
int e(int a, int b)
{
  while (b) {
    int q = b;
    b = a % b;
    a = q;
  }

  return a;
}
```

1. `1`
2. `7`
3. Infinite loop
4. `0`

Answer: `1`.

Obfuscation Quiz 1: MATH1081

What does this function return if called as `e(47, 19)`?

```
int e(int a, int b)
{
  while (b) {
    int q = b;
    b = a % b;
    a = q;
  }

  return a;
}
```

1. `1`
2. `7`
3. Infinite loop
4. `0`

Answer: `1`. This is actually the *Euclidean Algorithm* for calculating the GCD of two numbers. 47 and 19 are both prime, so their GCD is definitely 1.

Obfuscation Quiz 1: MATH1081

What does this function return if called as `e(47, 19)`?

```
int e(int a, int b)
{
  while (b) {
    int q = b;
    b = a % b;
    a = q;
  }

  return a;
}
```

1. `1`
2. `7`
3. Infinite loop
4. `0`

Answer: `1`. This is actually the *Euclidean Algorithm* for calculating the GCD of two numbers. 47 and 19 are both prime, so their GCD is definitely 1. For a non-mathematician especially the Euclidean Algorithm would be hard to identify just by psuedocode, so removing the name makes it difficult to read.

Preprocessor Macros

Preprocessor Macros

Kind of like a `#define` function. These should be used like small functions, but since they work like `#define` they can be used for silly things.

Preprocessor Macros

Kind of like a `#define` function. These should be used like small functions, but since they work like `#define` they can be used for silly things.

```
// Simple to read

#define FACTOR 42
int eliminate_divisible(int * numbers,
                        int size)
{
  int count = 0;

  for (int i = 0; i < size; i++) {
    if (numbers[i] % FACTOR == 0) {
      numbers[i] = 0;

      // Record how many numbers we
      // eliminate
      count++;
    }
  }

  return count;
}
```

Preprocessor Macros

Kind of like a `#define` function. These should be used like small functions, but since they work like `#define` they can be used for silly things.

```c
// Simple to read

#define FACTOR 42
int eliminate_divisible(int * numbers,
                        int size)
{
  int count = 0;

  for (int i = 0; i < size; i++) {
    if (numbers[i] % FACTOR == 0) {
      numbers[i] = 0;

      // Record how many numbers we
      // eliminate
      count++;
    }
  }

  return count;
}
```

```c
// Those look like functions,
// but are they really?

#define D(a,b,c) a[b] % c == 0
#define F(v,l) int v = 0; v < l; v++

int e(int * n, int s)
{
  int c = 0;

  for (F(i, s)) {
    if (D(n, i, 42)) {
      n[i] = 0;
      c++;
    }
  }

  return c;
}
```

Preprocessor Macros

Preprocessor Macros

Be warned, because these act just like regular `#defines`, they can lead to some strange unintended behaviour.

Preprocessor Macros

Be warned, because these act just like regular `#defines`, they can lead to some strange unintended behaviour.
This function *always* exits the program, regardless of whether or not there was an error.

Preprocessor Macros

Be warned, because these act just like regular `#define`s, they can lead to some strange unintended behaviour.
This function *always* exits the program, regardless of whether or not there was an error.

```
#define ERR(s) fprintf(stderr, s); exit(1);

FILE * open_check(const char * name)
{
  FILE * f = fopen(name, "r");

  if (f == NULL)
    ERR("Failed to open file\n");

  return f;
}
```

Preprocessor Macros

Be warned, because these act just like regular `#define`s, they can lead to some strange unintended behaviour.
This function *always* exits the program, regardless of whether or not there was an error.

```
#define ERR(s) fprintf(stderr, s); exit(1);

FILE * open_check(const char * name)
{
  FILE * f = fopen(name, "r");

  if (f == NULL)
    ERR("Failed to open file\n");

  return f;
}
```

How many semicolons are on the end of the line inside the `if` statement? Does this matter?

Obfuscation Quiz 2: Familiar Final Question

Obfuscation Quiz 2: Familiar Final Question

What does this program print?

```
#include <stdio.h>

#define A 3
#define B 5
#define C A + B

int main(void)
{
  printf("%d\n", C * A);
}
```

Obfuscation Quiz 2: Familiar Final Question

What does this program print?

```
#include <stdio.h>

#define A 3
#define B 5
#define C A + B

int main(void)
{
  printf("%d\n", C * A);
}
```

1. 24

Obfuscation Quiz 2: Familiar Final Question

What does this program print?

```
#include <stdio.h>

#define A 3
#define B 5
#define C A + B

int main(void)
{
  printf("%d\n", C * A);
}
```

1. 24
2. Compilation error

Obfuscation Quiz 2: Familiar Final Question

What does this program print?

```
#include <stdio.h>

#define A 3
#define B 5
#define C A + B

int main(void)
{
  printf("%d\n", C * A);
}
```

1. 24
2. Compilation error
3. Undefined behaviour

Obfuscation Quiz 2: Familiar Final Question

What does this program print?

```
#include <stdio.h>

#define A 3
#define B 5
#define C A + B

int main(void)
{
  printf("%d\n", C * A);
}
```

1. 24
2. Compilation error
3. Undefined behaviour
4. 18

Obfuscation Quiz 2: Familiar Final Question

What does this program print?

```
#include <stdio.h>

#define A 3
#define B 5
#define C A + B

int main(void)
{
  printf("%d\n", C * A);
}
```

1. 24
2. Compilation error
3. Undefined behaviour
4. 18

Answer: 18.

Obfuscation Quiz 2: Familiar Final Question

What does this program print?

```
#include <stdio.h>

#define A 3
#define B 5
#define C A + B

int main(void)
{
  printf("%d\n", C * A);
}
```

1. 24
2. Compilation error
3. Undefined behaviour
4. 18

Answer: `18`. The macros are expanded literally to be `3 + 5 * 3`, where BIDMAS takes over.

Obfuscation Quiz 2: Familiar Final Question

What does this program print?

```
#include <stdio.h>

#define A 3
#define B 5
#define C A + B

int main(void)
{
  printf("%d\n", C * A);
}
```

1. 24
2. Compilation error
3. Undefined behaviour
4. 18

Answer: 18. The macros are expanded literally to be `3 + 5 * 3`, where BIDMAS takes over. This is in contrast to what would happen if `A`, `B` and `C` were variables.

Caveats

Caveats

Using these things by themselves has problems.

Caveats

Using these things by themselves has problems.

- If people analyse your source code, they can make their own comments and give variables their own names, and it's easier to analyse.

Caveats

Using these things by themselves has problems.

- If people analyse your source code, they can make their own comments and give variables their own names, and it's easier to analyse.
- Doesn't work well for simple functions, sometimes people write simple functions like that anyway.

Caveats

Using these things by themselves has problems.

- If people analyse your source code, they can make their own comments and give variables their own names, and it's easier to analyse.
- Doesn't work well for simple functions, sometimes people write simple functions like that anyway.
- Macros can be defeated trivially with `gcc -E`.

Caveats

Using these things by themselves has problems.

- If people analyse your source code, they can make their own comments and give variables their own names, and it's easier to analyse.
- Doesn't work well for simple functions, sometimes people write simple functions like that anyway.
- Macros can be defeated trivially with `gcc -E`.
- Macros also must be on their own line (more relevant for golfing which we will discuss later).

Contents

Ternary Operators

Ternary Operators

Simple `if` statements can be replaced with the ternary operator:

Ternary Operators

Simple `if` statements can be replaced with the ternary operator:

```
// A function using if statements

char toggle_case(char c)
{
  if (c >= 'a' && c <= 'z') {
    return c + ('A' - 'a');
  } else {
    return c + ('a' - 'A');
  }
}
```

Ternary Operators

Simple `if` statements can be replaced with the ternary operator:

```
// A function using if statements

char toggle_case(char c)
{
  if (c >= 'a' && c <= 'z') {
    return c + ('A' - 'a');
  } else {
    return c + ('a' - 'A');
  }
}
```

```
// A function using the ternary operator

char toggle_case(char c)
{
  return (c >= 'a' && c <= 'z')
         ? c + ('A' - 'a')
         : c + ('a' - 'A');
}
```

Ternary Operators

Ternary Operators

Because a ternary operator takes the place of a *value* rather than a *statement* (like an `if` statement), it can be used for some sneaky tricks:

Ternary Operators

Because a ternary operator takes the place of a *value* rather than a *statement* (like an `if` statement), it can be used for some sneaky tricks:

```
int factors(int n)
{
  int c = 0;

  for (int i = 1; i <= n; i++) {
    // What's happening here?
    c += (n % i) ? 0 : 1;
  }

  return c;
}
```

Ternary Operators

Because a ternary operator takes the place of a *value* rather than a *statement* (like an `if` statement), it can be used for some sneaky tricks:

```
int factors(int n)
{
  int c = 0;

  for (int i = 1; i <= n; i++) {
    // What's happening here?
    c += (n % i) ? 0 : 1;
  }

  return c;
}
```

```
void fizzbuzz(int n)
{
  // Ace your interviews with this one
  // weird trick!
  printf((n % 3 && n % 5)
         ? "%d%s\n"
         : "%s%s\n",
         (n % 3 && n % 5)
         ? n
         : ((n % 3) ? "" : "Fizz"),
         (n % 5)
         ? ""
         : "Buzz");
}
```

As the example on the right shows, nesting ternary operators can lead to some very nasty code.

Obfuscation Quiz 3: Positives and Negatives

Obfuscation Quiz 3: Positives and Negatives

What does this code print?

```
#include <stdio.h>

int main(void)
{
  int a = 0, b = 0, c = 0;
  printf("%d\n", (a=1)?(a--?(--a+(++b)+c++):(c+++a)):(--a-(--b)-(--c)));
}
```

Obfuscation Quiz 3: Positives and Negatives

What does this code print?

```
#include <stdio.h>

int main(void)
{
  int a = 0, b = 0, c = 0;
  printf("%d\n", (a=1)?(a--?(--a+(++b)+c++):(c+++a)):(--a-(--b)-(--c)));
}
```

1. 1

Obfuscation Quiz 3: Positives and Negatives

What does this code print?

```c
#include <stdio.h>

int main(void)
{
  int a = 0, b = 0, c = 0;
  printf("%d\n", (a=1)?(a--?(--a+(++b)+c++):(c+++a)):(--a-(--b)-(--c)));
}
```

1. 1
2. 0

Obfuscation Quiz 3: Positives and Negatives

What does this code print?

```
#include <stdio.h>

int main(void)
{
  int a = 0, b = 0, c = 0;
  printf("%d\n", (a=1)?(a--?(--a+(++b)+c++):(c+++a)):(--a-(--b)-(--c)));
}
```

1. 1
2. 0
3. -1

Obfuscation Quiz 3: Positives and Negatives

What does this code print?

```
#include <stdio.h>

int main(void)
{
  int a = 0, b = 0, c = 0;
  printf("%d\n", (a=1)?(a--?(--a+(++b)+c++):(c+++a)):(--a-(--b)-(--c)));
}
```

1. 1
2. 0
3. -1
4. Compilation error

Obfuscation Quiz 3: Positives and Negatives

What does this code print?

```
#include <stdio.h>

int main(void)
{
  int a = 0, b = 0, c = 0;
  printf("%d\n", (a=1)?(a--?(--a+(++b)+c++):(c+++a)):(--a-(--b)-(--c)));
}
```

1. 1
2. 0
3. -1
4. Compilation error

Answer: `0`.

Obfuscation Quiz 3: Positives and Negatives

What does this code print?

```
#include <stdio.h>

int main(void)
{
  int a = 0, b = 0, c = 0;
  printf("%d\n", (a=1)?(a--?(--a+(++b)+c++):(c+++a)):(--a-(--b)-(--c)));
}
```

1. 1
2. 0
3. -1
4. Compilation error

Answer: `0`. `a = 1` sets `a` to `1` and evaluates as `1` (true).

Obfuscation Quiz 3: Positives and Negatives

What does this code print?

```
#include <stdio.h>

int main(void)
{
  int a = 0, b = 0, c = 0;
  printf("%d\n", (a=1)?(a--?(--a+(++b)+c++):(c+++a)):(--a-(--b)-(--c)));
}
```

1. 1
2. 0
3. -1
4. Compilation error

Answer: `0`. `a = 1` sets `a` to `1` and evaluates as `1` (true). `a--` sets `a` to `0` but evaluates as `1` (true).

Obfuscation Quiz 3: Positives and Negatives

What does this code print?

```c
#include <stdio.h>

int main(void)
{
  int a = 0, b = 0, c = 0;
  printf("%d\n", (a=1)?(a--?(--a+(++b)+c++):(c+++a)):(--a-(--b)-(--c)));
}
```

1. 1
2. 0
3. -1
4. Compilation error

Answer: `0`. `a = 1` sets `a` to `1` and evaluates as `1` (true). `a--` sets `a` to `0` but evaluates as `1` (true). Then `--a+(++b)+c++` evaluates to `--0+(++0)+0++` or `(-1)+(1)+(0)` (but setting `c` to 1 after evaluating).

Obfuscation Quiz 3: Positives and Negatives

What does this code print?

```c
#include <stdio.h>

int main(void)
{
  int a = 0, b = 0, c = 0;
  printf("%d\n", (a=1)?(a--?(--a+(++b)+c++):(c+++a)):(--a-(--b)-(--c)));
}
```

1. 1
2. 0
3. -1
4. Compilation error

Answer: `0`. `a = 1` sets `a` to `1` and evaluates as `1` (true). `a--` sets `a` to `0` but evaluates as `1` (true). Then `--a+(++b)+c++` evaluates to `--0+(++0)+0++` or `(-1)+(1)+(0)` (but setting `c` to 1 after evaluating). Side note: Is `c+++a` equivalent to `(c++)+a` or `c+(++a)`?

Conditionals in Arithmetic

Conditionals in Arithmetic

In C, zero is treated as false, and non-zero values are treated as true. You've probably seen this being used already to check for the end of strings, or the return value of `malloc()`.

Conditionals in Arithmetic

In C, zero is treated as false, and non-zero values are treated as true. You've probably seen this being used already to check for the end of strings, or the return value of `malloc()`.

```c
// String length

int length(char * str)
{
  int l = 0;

  for (int i = 0; s[i]; i++) {
    l++;
  }

  return l;
}
```

Conditionals in Arithmetic

In C, zero is treated as false, and non-zero values are treated as true. You've probably seen this being used already to check for the end of strings, or the return value of `malloc()`.

```
// String length

int length(char * str)
{
  int l = 0;

  for (int i = 0; s[i]; i++) {
    l++;
  }

  return l;
}
```

```
// New linked list node

node_t * new_node(int val)
{
  node_t * new = malloc(sizeof(node_t));

  if (!new) {
    fprintf(stderr, "PANIC! PANIC!\n");
    return NULL;
  }

  new->next = NULL;
  new->val = val;

  return new;
}
```

Conditionals in Arithmetic

Conditionals in Arithmetic

This trick lets us write even shorter / weirder versions of our ternary operator statements in some cases.

Conditionals in Arithmetic

This trick lets us write even shorter / weirder versions of our ternary operator statements in some cases.

```
int factors(int n)
{
  int c = 0;

  for (int i = 1; i <= n; i++) {
    c += !(n % i);
  }

  return c;
}
```

Conditionals in Arithmetic

This trick lets us write even shorter / weirder versions of our ternary operator statements in some cases.

```
int factors(int n)
{
  int c = 0;

  for (int i = 1; i <= n; i++) {
    c += !(n % i);
  }

  return c;
}
```

```
double median(double * x, int n)
{
  return (x[n / 2]
         + (!(n % 2) * x[n / 2 - 1]))
         / (!(n % 2) + 1);
}
```

Conditionals in Arithmetic

Conditionals in Arithmetic

Be warned, unlike `if` statements or the ternary operator, conditionals in arithmetic do not stop the other case from being *evaluated*, they can only stop it from being used.

Conditionals in Arithmetic

Be warned, unlike `if` statements or the ternary operator, conditionals in arithmetic do not stop the other case from being *evaluated*, they can only stop it from being used.

```
// Stack overflow!

unsigned int factorial(unsigned int n) {
  return n * (factorial(n - 1) - 1) + 1;
}
```

Obfuscation Quiz 4: That Sequence

Obfuscation Quiz 4: That Sequence

What does this code print?

```
int main(void)
{
  int n = 5;

  do {
    printf("%d\n", n);
  } while ((n = (1 - (n % 2)) * (n / 2) + (n % 2) * (3 * n + 1)) - 1);

  printf("%d\n", n);
}
```

Obfuscation Quiz 4: That Sequence

What does this code print?

```
int main(void)
{
  int n = 5;

  do {
    printf("%d\n", n);
  } while ((n = (1 - (n % 2)) * (n / 2) + (n % 2) * (3 * n + 1)) - 1);

  printf("%d\n", n);
}
```

1. 5, 5, 5, 5 ... (Infinite loop)

Obfuscation Quiz 4: That Sequence

What does this code print?

```
int main(void)
{
  int n = 5;

  do {
    printf("%d\n", n);
  } while ((n = (1 - (n % 2)) * (n / 2) + (n % 2) * (3 * n + 1)) - 1);

  printf("%d\n", n);
}
```

1. 5, 5, 5, 5 ... (Infinite loop)
2. 5, 16, 8, 4, 2, 1

Obfuscation Quiz 4: That Sequence

What does this code print?

```
int main(void)
{
  int n = 5;

  do {
    printf("%d\n", n);
  } while ((n = (1 - (n % 2)) * (n / 2) + (n % 2) * (3 * n + 1)) - 1);

  printf("%d\n", n);
}
```

1. 5, 5, 5, 5 ... (Infinite loop)
2. 5, 16, 8, 4, 2, 1
3. Random infinite sequence

Obfuscation Quiz 4: That Sequence

What does this code print?

```c
int main(void)
{
  int n = 5;

  do {
    printf("%d\n", n);
  } while ((n = (1 - (n % 2)) * (n / 2) + (n % 2) * (3 * n + 1)) - 1);

  printf("%d\n", n);
}
```

1. 5, 5, 5, 5 ... (Infinite loop)
2. 5, 16, 8, 4, 2, 1
3. Random infinite sequence
4. 5

Obfuscation Quiz 4: That Sequence

What does this code print?

```
int main(void)
{
  int n = 5;

  do {
    printf("%d\n", n);
  } while ((n = (1 - (n % 2)) * (n / 2) + (n % 2) * (3 * n + 1)) - 1);

  printf("%d\n", n);
}
```

1. 5, 5, 5, 5 ... (Infinite loop)
2. 5, 16, 8, 4, 2, 1
3. Random infinite sequence
4. 5

Answer: 5, 16, 8, 4, 2, 1.

Obfuscation Quiz 4: That Sequence

What does this code print?

```
int main(void)
{
  int n = 5;

  do {
    printf("%d\n", n);
  } while ((n = (1 - (n % 2)) * (n / 2) + (n % 2) * (3 * n + 1)) - 1);

  printf("%d\n", n);
}
```

1. 5, 5, 5, 5 ... (Infinite loop)
2. 5, 16, 8, 4, 2, 1
3. Random infinite sequence
4. 5

Answer: 5, 16, 8, 4, 2, 1. This is the *Hailstone Sequence* from the somewhat well-known *Collatz Conjecture*.

Obfuscation Quiz 4: That Sequence

What does this code print?

```
int main(void)
{
  int n = 5;

  do {
    printf("%d\n", n);
  } while ((n = (1 - (n % 2)) * (n / 2) + (n % 2) * (3 * n + 1)) - 1);

  printf("%d\n", n);
}
```

1. 5, 5, 5, 5 ... (Infinite loop)
2. 5, 16, 8, 4, 2, 1
3. Random infinite sequence
4. 5

Answer: 5, 16, 8, 4, 2, 1. This is the *Hailstone Sequence* from the somewhat well-known *Collatz Conjecture*. The number will eventually return to 1, stopping the loop since `(n = ...) - 1` will evaluate to `(1) - 1`.

goto — The Most Evil Keyword

`goto` — The Most Evil Keyword

C programmers are told to never use `goto`, because when misused it makes the program very difficult to follow. Fortunately, this is very interesting to us!

goto — The Most Evil Keyword

C programmers are told to never use `goto`, because when misused it makes the program very difficult to follow. Fortunately, this is very interesting to us!

`goto` is also a clean and well accepted way to do error handling in long functions if you have some complex error handling.

```
big_struct_t * initialise()
{
  big_struct_t * new = malloc(sizeof(big_struct_t));
  if (new == NULL)
    goto error;

  // More initialisation which might fail...

  return new;

error:
  fprintf(stderr, "Could not create big_struct_t\n");

  // More complex error handling...

  return NULL;
}
```

`goto` — The Most Evil Keyword

goto — The Most Evil Keyword

With enough labels, you can make your code go in whatever order you want:

goto — The Most Evil Keyword

With enough labels, you can make your code go in whatever order you want:

```
// Easy(ish) to follow

int palindrome_factor(char * str)
{
  int len = strlen(str);
  char * tmp = malloc(len + 1);

  for (int i = len, j = 0;
       i >= 0;
       i--, j++) {
    tmp[i] = str[j];
  }

  int count = 0;

  for (int i = 0; i < len; i++) {
    if (tmp[i] == str[i])
      count++;
  }

  return count;
}
```

goto — The Most Evil Keyword

With enough labels, you can make your code go in whatever order you want:

```c
// Easy(ish) to follow

int palindrome_factor(char * str)
{
  int len = strlen(str);
  char * tmp = malloc(len + 1);

  for (int i = len, j = 0;
       i >= 0;
       i--, j++) {
    tmp[i] = str[j];
  }

  int count = 0;

  for (int i = 0; i < len; i++) {
    if (tmp[i] == str[i])
      count++;
  }

  return count;
}
```

```c
// Hard to follow

int palindrome_factor(char * str)
{
  int len; char * tmp; int count;
  int i; int j;
  goto d;

a:
  i = 0; count = 0; goto e;
b:
  tmp[i--] = str[j++];
  if (i >= 0) goto b; goto a;
c:
  i = len - 1, j = 0; goto b;
d:
  len = strlen(str); tmp = malloc(len + 1);
  goto c;
e:
  if (tmp[i] == str[i]) count++; i++;
  if (i < len) goto e; return count;
}
```

The Almighty `for`

The Almighty `for`

- What is a loop generally?

The Almighty `for`

- What is a loop generally?
  1. Initialise some variables.

The Almighty `for`

- What is a loop generally?
  1. Initialise some variables.
  2. Do some operation.

The Almighty `for`

- What is a loop generally?
  1. Initialise some variables.
  2. Do some operation.
  3. Check some condition.

The Almighty `for`

- What is a loop generally?
  1. Initialise some variables.
  2. Do some operation.
  3. Check some condition.
  4. If the condition is true, repeat.

The Almighty `for`

- What is a loop generally?
  1. Initialise some variables.
  2. Do some operation.
  3. Check some condition.
  4. If the condition is true, repeat.
- What does the `for` keyword let us do?

The Almighty `for`

- What is a loop generally?
  1. Initialise some variables.
  2. Do some operation.
  3. Check some condition.
  4. If the condition is true, repeat.
- What does the `for` keyword let us do?
  1. Run one statement once (initialisation).

The Almighty `for`

- What is a loop generally?
  1. Initialise some variables.
  2. Do some operation.
  3. Check some condition.
  4. If the condition is true, repeat.
- What does the `for` keyword let us do?
  1. Run one statement once (initialisation).
  2. Run one *block* repetitively (loop body).

The Almighty `for`

- What is a loop generally?
  1. Initialise some variables.
  2. Do some operation.
  3. Check some condition.
  4. If the condition is true, repeat.
- What does the `for` keyword let us do?
  1. Run one statement once (initialisation).
  2. Run one *block* repetitively (loop body).
  3. Run one statement after that block (usually incrementing).

The Almighty `for`

- What is a loop generally?
  1. Initialise some variables.
  2. Do some operation.
  3. Check some condition.
  4. If the condition is true, repeat.
- What does the `for` keyword let us do?
  1. Run one statement once (initialisation).
  2. Run one *block* repetitively (loop body).
  3. Run one statement after that block (usually incrementing).
  4. Check one condition before running the block, stop if false (condition).

The Almighty `for`

- What is a loop generally?
  1. Initialise some variables.
  2. Do some operation.
  3. Check some condition.
  4. If the condition is true, repeat.
- What does the `for` keyword let us do?
  1. Run one statement once (initialisation).
  2. Run one *block* repetitively (loop body).
  3. Run one statement after that block (usually incrementing).
  4. Check one condition before running the block, stop if false (condition).
- `for` loops are surprisingly general.

The Almighty `for`

The Almighty `for`

`while` loops are quite general too, but because `for` loops are usually used for one specific type of loop, it's more jarring to see them used for something else.

The Almighty for

`while` loops are quite general too, but because `for` loops are usually used for one specific type of loop, it's more jarring to see them used for something else. How jarring is this code?

```
int sum_list(node_t * list)
{
  int sum = 0;

  // Shouldn't this normally be a while loop?
  for (node_t * curr = list; curr != NULL; curr = curr->next) {
    sum += curr->val;
  }

  return sum;
}
```

The Almighty `for`

The Almighty `for`

Common programming structures like `if` and `while` can be converted into a `for` “loop” fairly easily:

The Almighty for

Common programming structures like `if` and `while` can be converted into a `for` "loop" fairly easily:

```
if (condition()) {
  action();
}
```

The Almighty `for`

Common programming structures like `if` and `while` can be converted into a `for` "loop" fairly easily:

```
if (condition()) {
  action();
}
```

```
for (int i = condition(); i; i = 0) {
  action();
}

// Alternatively:

for (;condition();) {
  action();
  break;
}
```

The Almighty for

Common programming structures like `if` and `while` can be converted into a `for` “loop” fairly easily:

```
if (condition()) {
  action();
}
```

```
for (int i = condition(); i; i = 0) {
  action();
}

// Alternatively:

for (;condition();) {
  action();
  break;
}
```

```
while (condition()) {
  action();
}
```

The Almighty for

Common programming structures like `if` and `while` can be converted into a `for` "loop" fairly easily:

```
if (condition()) {
  action();
}
```

```
for (int i = condition(); i; i = 0) {
  action();
}

// Alternatively:

for (;condition();) {
  action();
  break;
}
```

```
while (condition()) {
  action();
}
```

```
for (;condition();) {
  action();
}
```

The Almighty `for`

The Almighty `for`

With a bit of trickery and the comma “operator”, nested `for` loops or successive `for` loops can be turned into a single `for` loop:

The Almighty for

With a bit of trickery and the comma "operator", nested `for` loops or successive `for` loops can be turned into a single `for` loop:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    action(i, j);
  }
}
```

The Almighty for

With a bit of trickery and the comma “operator”, nested for loops or successive for loops can be turned into a single for loop:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    action(i, j);
  }
}
```

```
for (int i = 0, j = 0;
     i < n;
     j = (j + 1) % m, i += (j == 0)) {
  action(i, j);
}
```

The Almighty for

With a bit of trickery and the comma "operator", nested for loops or successive for loops can be turned into a single for loop:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    action(i, j);
  }
}
```

```
for (int i = 0, j = 0;
     i < n;
     j = (j + 1) % m, i += (j == 0)) {
  action(i, j);
}
```

```
for (int i = 0; i < n; i++) {
  action(i);
}

for (int j = 0; i < m; j++) {
  other_action(j);
}
```

The Almighty `for`

With a bit of trickery and the comma "operator", nested `for` loops or successive `for` loops can be turned into a single `for` loop:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    action(i, j);
  }
}
```

```
for (int i = 0, j = 0;
     i < n;
     j = (j + 1) % m, i += (j == 0)) {
  action(i, j);
}
```

```
for (int i = 0; i < n; i++) {
  action(i);
}

for (int j = 0; i < m; j++) {
  other_action(j);
}
```

```
// This does use an if statement,
// but it's still harder to read

for (int i = 0, j = 0, k = 0;
     k <= 2;
     i += (k == 0), j += (k == 1),
     k += (i == n || j == m)) {
  if (k == 0)
    action(i);
  else
    action(j);
}
```

The Almighty `for`

The Almighty `for`

An even smaller version exists for certain kinds of nested loops, particularly "square" ones, which are useful for dealing with graphics.

The Almighty `for`

An even smaller version exists for certain kinds of nested loops, particularly "square" ones, which are useful for dealing with graphics.

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    action(i, j);
  }
}
```

The Almighty `for`

An even smaller version exists for certain kinds of nested loops, particularly "square" ones, which are useful for dealing with graphics.

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    action(i, j);
  }
}
```

```
// Take note that the *outer* loop uses /
// and the *inner* loop uses %

for (int i = 0; i < n * n; i++) {
  action(i / n, i % n);
}
```

The Almighty `for`

The Almighty `for`

The “incrementation” part of the for loop is essentially just a compulsory line at the end of the `for` loop body. Thanks to the comma “operator”, you can turn smaller `for` loops into “empty” ones:

The Almighty for

The "incrementation" part of the for loop is essentially just a compulsory line at the end of the `for` loop body. Thanks to the comma "operator", you can turn smaller `for` loops into "empty" ones:

```
for (int i = 0; i < n; i++) {
  action(i);
}
```

The Almighty `for`

The “incrementation” part of the for loop is essentially just a compulsory line at the end of the `for` loop body. Thanks to the comma “operator”, you can turn smaller `for` loops into “empty” ones:

```
for (int i = 0; i < n; i++) {
  action(i);
}
```

```
// We don't actually use the comma operator
// but you might need to

for (int i = 0; i < n; action(i++));
```

Obfuscation Quiz 5: Genetic Algorithm?

Obfuscation Quiz 5: Genetic Algorithm?

What sort of pattern does this code print?

```
for (int i = 0, j = 0, k = 0; i < 200; j = (++i) / 10 % 10, k = i % 10) {
  if (k == j) {
    printf("\\");
  } else if (k == 9 - j) {
    printf("/");
  } else if ((k >= j && k <= 9 - j) || (k <= j && k >= 9 - j)) {
    printf("~");
  } else {
    printf(" ");
  }

  if (k == 9)
    printf("\n");
}
```

Obfuscation Quiz 5: Genetic Algorithm?

What sort of pattern does this code print?

```
for (int i = 0, j = 0, k = 0; i < 200; j = (++i) / 10 % 10, k = i % 10) {
  if (k == j) {
    printf("\\");
  } else if (k == 9 - j) {
    printf("/");
  } else if ((k >= j && k <= 9 - j) || (k <= j && k >= 9 - j)) {
    printf("~");
  } else {
    printf(" ");
  }

  if (k == 9)
    printf("\n");
}
```

1. Spiral

Obfuscation Quiz 5: Genetic Algorithm?

What sort of pattern does this code print?

```
for (int i = 0, j = 0, k = 0; i < 200; j = (++i) / 10 % 10, k = i % 10) {
  if (k == j) {
    printf("\\");
  } else if (k == 9 - j) {
    printf("/");
  } else if ((k >= j && k <= 9 - j) || (k <= j && k >= 9 - j)) {
    printf("~");
  } else {
    printf(" ");
  }

  if (k == 9)
    printf("\n");
}
```

1. Spiral
2. Box

Obfuscation Quiz 5: Genetic Algorithm?

What sort of pattern does this code print?

```
for (int i = 0, j = 0, k = 0; i < 200; j = (++i) / 10 % 10, k = i % 10) {
  if (k == j) {
    printf("\\");
  } else if (k == 9 - j) {
    printf("/");
  } else if ((k >= j && k <= 9 - j) || (k <= j && k >= 9 - j)) {
    printf("~");
  } else {
    printf(" ");
  }

  if (k == 9)
    printf("\n");
}
```

1. Spiral
2. Box
3. Grid

Obfuscation Quiz 5: Genetic Algorithm?

What sort of pattern does this code print?

```
for (int i = 0, j = 0, k = 0; i < 200; j = (++i) / 10 % 10, k = i % 10) {
  if (k == j) {
    printf("\\");
  } else if (k == 9 - j) {
    printf("/");
  } else if ((k >= j && k <= 9 - j) || (k <= j && k >= 9 - j)) {
    printf("~");
  } else {
    printf(" ");
  }

  if (k == 9)
    printf("\n");
}
```

1. Spiral
2. Box
3. Grid
4. Helix

Obfuscation Quiz 5: Genetic Algorithm?

What sort of pattern does this code print?

```
for (int i = 0, j = 0, k = 0; i < 200; j = (++i) / 10 % 10, k = i % 10) {
  if (k == j) {
    printf("\\");
  } else if (k == 9 - j) {
    printf("/");
  } else if ((k >= j && k <= 9 - j) || (k <= j && k >= 9 - j)) {
    printf("~");
  } else {
    printf(" ");
  }

  if (k == 9)
    printf("\n");
}
```

1. Spiral
2. Box
3. Grid
4. Helix

Answer: Helix (specifically, a DNA-style helix).

Obfuscation Quiz 5: Genetic Algorithm?

What sort of pattern does this code print?

```
for (int i = 0, j = 0, k = 0; i < 200; j = (++i) / 10 % 10, k = i % 10) {
  if (k == j) {
    printf("\\");
  } else if (k == 9 - j) {
    printf("/");
  } else if ((k >= j && k <= 9 - j) || (k <= j && k >= 9 - j)) {
    printf("~");
  } else {
    printf(" ");
  }

  if (k == 9)
    printf("\n");
}
```

1. Spiral
2. Box
3. Grid
4. Helix

Answer: Helix (specifically, a DNA-style helix). The pattern isn't particularly difficult to follow since it's essentially a filled-in cross.

Obfuscation Quiz 5: Genetic Algorithm?

What sort of pattern does this code print?

```
for (int i = 0, j = 0, k = 0; i < 200; j = (++i) / 10 % 10, k = i % 10) {
  if (k == j) {
    printf("\\");
  } else if (k == 9 - j) {
    printf("/");
  } else if ((k >= j && k <= 9 - j) || (k <= j && k >= 9 - j)) {
    printf("~");
  } else {
    printf(" ");
  }

  if (k == 9)
    printf("\n");
}
```

1. Spiral
2. Box
3. Grid
4. Helix

Answer: Helix (specifically, a DNA-style helix). The pattern isn't particularly difficult to follow since it's essentially a filled-in cross. The difficulty comes from the structure of the `for` loops being lost.

Contents

Variable Economy

Variable Economy

Variables are simply a means to giving a name to a region of memory.

Variable Economy

Variables are simply a means to giving a name to a region of memory. A name helps give meaning — even if the name is just `a` — it signifies that this section of memory has some distinct purpose.

Variable Economy

Variables are simply a means to giving a name to a region of memory.
A name helps give meaning — even if the name is just `a` — it signifies that this section of memory has some distinct purpose.
We want to use as little variables as possible.

Variable Economy

Variables are simply a means to giving a name to a region of memory.
A name helps give meaning — even if the name is just `a` — it signifies that this section of memory has some distinct purpose.
We want to use as little variables as possible.

```
// Swap two variables without using a third
// Avoids using the XOR trick, but watch out for overflow!
a = a + b
b = a - b
a = a - b
```

Variable Economy

Variable Economy

- Strategy one:

Variable Economy

- Strategy one: Giant global array.

Variable Economy

- Strategy one: Giant global array.
- Rather than variables, simply have a global array with a pre-calculated size and with a good data type

Variable Economy

- Strategy one: Giant global array.
- Rather than variables, simply have a global array with a pre-calculated size and with a good data type (like `int` — for alignment purposes).

Variable Economy

- Strategy one: Giant global array.
- Rather than variables, simply have a global array with a pre-calculated size and with a good data type (like `int` — for alignment purposes).
- Use offsets into that array in place of variable names.

Variable Economy

- Strategy one: Giant global array.
- Rather than variables, simply have a global array with a pre-calculated size and with a good data type (like `int` — for alignment purposes).
- Use offsets into that array in place of variable names.
- Good for programs using multiple arrays.

Variable Economy

Variable Economy

```c
int sum_matching(int * a1, int * a2, int n)
{
  int sum = 0;

  for (int i = 0; i < n; i++) {
    sum += a1[i] + a2[i];
  }

  return sum;
}

int main(int argc, char ** argv)
{
  int n = atoi(argv[1]);
  int * a1 = malloc(sizeof(int) * n);
  int * a2 = malloc(sizeof(int) * n);

  for (int i = 0; i < n; i++)
    a1[i] = atoi(argv[i + 2]);
  for (int i = 0; i < n; i++)
    a2[i] = atoi(argv[i + 2 + n]);

  printf("%d\n", sum_matching(a1, a2, n));
  return 0;
}
```

Variable Economy

```c
int sum_matching(int * a1, int * a2, int n)
{
  int sum = 0;

  for (int i = 0; i < n; i++) {
    sum += a1[i] + a2[i];
  }

  return sum;
}

int main(int argc, char ** argv)
{
  int n = atoi(argv[1]);
  int * a1 = malloc(sizeof(int) * n);
  int * a2 = malloc(sizeof(int) * n);

  for (int i = 0; i < n; i++)
    a1[i] = atoi(argv[i + 2]);
  for (int i = 0; i < n; i++)
    a2[i] = atoi(argv[i + 2 + n]);

  printf("%d\n", sum_matching(a1, a2, n));
  return 0;
}
```

```c
int data[2003];

void sum_matching(void)
{
  for (*data = 0; *data < data[1];
       (*data)++) {
    data[2] += data[3 + *data]
             + data[1003 + *data];
  }
}

int main(int argc, char ** argv)
{
  data[1] = atoi(argv[1]);

  for (; *data < data[1]; (*data)++)
    data[3 + *data]
    = atoi(argv[*data + 2]);
  for (*data = 0; *data < data[1];
       (*data)++)
    data[1003 + *data]
    = atoi(argv[*data + 2 + data[1]]);

  sum_matching();
  printf("%d\n", data[2]);
  return 0;
}
```

Variable Economy

Variable Economy

- Strategy two:

Variable Economy

- Strategy two: Variable re-use.

Variable Economy

- Strategy two: Variable re-use.
- Most variables are only used for some portion of their scope.

Variable Economy

- Strategy two: Variable re-use.
- Most variables are only used for some portion of their scope.
- To reduce the number of variables we use, we can reuse them.

Variable Economy

- Strategy two: Variable re-use.
- Most variables are only used for some portion of their scope.
- To reduce the number of variables we use, we can reuse them.
- Prime candidates:

Variable Economy

- Strategy two: Variable re-use.
- Most variables are only used for some portion of their scope.
- To reduce the number of variables we use, we can reuse them.
- Prime candidates:
    - `argc` and `argv`

Variable Economy

- Strategy two: Variable re-use.
- Most variables are only used for some portion of their scope.
- To reduce the number of variables we use, we can reuse them.
- Prime candidates:
  - `argc` and `argv`
  - `for` loop counters.

Variable Economy

- Strategy two: Variable re-use.
- Most variables are only used for some portion of their scope.
- To reduce the number of variables we use, we can reuse them.
- Prime candidates:
    - `argc` and `argv`
    - `for` loop counters.
    - Temporary variables.

Variable Economy

- Strategy two: Variable re-use.
- Most variables are only used for some portion of their scope.
- To reduce the number of variables we use, we can reuse them.
- Prime candidates:
  - `argc` and `argv`
  - `for` loop counters.
  - Temporary variables.
- It really helps to have all your variables be global.

Array Switches

Array Switches

Sometimes, you may be able to replace long `if-else` chains with just one array and some clever indexing.

Array Switches

Sometimes, you may be able to replace long `if-else` chains with just one array and some clever indexing.

```
void fizzbuzz(int n)
{
        if (n % 15 == 0) {
                printf("FizzBuzz\n");
        } else if (n % 5 == 0) {
                printf("Buzz\n");
        } else if (n % 3 == 0) {
                printf("Fizz\n");
        } else {
                printf("%d\n", n);
        }

        return 0;
}
```

Array Switches

Sometimes, you may be able to replace long `if-else` chains with just one array and some clever indexing.

```
void fizzbuzz(int n)
{
        if (n % 15 == 0) {
                printf("FizzBuzz\n");
        } else if (n % 5 == 0) {
                printf("Buzz\n");
        } else if (n % 3 == 0) {
                printf("Fizz\n");
        } else {
                printf("%d\n", n);
        }

        return 0;
}
```

```
char * fmt[4] = { "%d\n",
                  "Fizz\n",
                  "Buzz\n",
                  "FizzBuzz\n" };

void fizzbuzz(int n)
{
        printf(fmt[(n % 3 == 0)
                   + 2 * (n % 5 == 0)], n);
        return 0;
}
```

Confusing Arithmetic

Confusing Arithmetic

It's time to put your maths skills and your COMP1521 bitwise magic skills to good use!

Confusing Arithmetic

It’s time to put your maths skills and your COMP1521 bitwise magic skills to good use! Rewrite simple expressions as more complicated ones which are mathematically equivalent.

Confusing Arithmetic

It's time to put your maths skills and your COMP1521 bitwise magic skills to good use! Rewrite simple expressions as more complicated ones which are mathematically equivalent.

```c
int count_bits(unsigned int n)
{
  int count = 0;

  for (int i = 0; i < 32; i++) {
    if (n & (1 << i))
      count++;
  }

  return count;
}
```

Confusing Arithmetic

It's time to put your maths skills and your COMP1521 bitwise magic skills to good use! Rewrite simple expressions as more complicated ones which are mathematically equivalent.

```
int count_bits(unsigned int n)
{
  int count = 0;

  for (int i = 0; i < 32; i++) {
    if (n & (1 << i))
      count++;
  }

  return count;
}
```

```
int count_bits(unsigned int n)
{
  int count = 0;

  // A famous trick
  // Think about why it works
  while (n > 0) {
    n &= n - 1;
    count++;
  }

  return count;
}
```

Confusing Arithmetic

Confusing Arithmetic

Bitfields are often harder to read than just a regular array of bools, despite accomplishing the same thing.

Confusing Arithmetic

Bitfields are often harder to read than just a regular array of bools, despite accomplishing the same thing.

```
int flag1 = 0, flag2 = 0, flag3 = 0;

flag2 = 1;

if ((flag1 || flag2) && flag3) {
  do_something();
}
```

Confusing Arithmetic

Bitfields are often harder to read than just a regular array of bools, despite accomplishing the same thing.

```
int flag1 = 0, flag2 = 0, flag3 = 0;

flag2 = 1;

if ((flag1 || flag2) && flag3) {
  do_something();
}
```

```
int field = 0;

field |= 1 << 1;

if ((field & 3) && (field & 4)) {
  do_something();
}
```

Weird Literals

Weird Literals

String literals are essentially just a statically defined `char` array, and so you can use them to define such an array easily.

Quiz 6: Tribute to Grothendieck

Quiz 6: Tribute to Grothendieck

What does this code print?

Quiz 6: Tribute to Grothendieck

What does this code print?

```
char* p = "\2\3\5\7\v\r\x11\x13\x17\x1d\x1f%)+/5;=CGIOSYa";

int main(void)
{
  int n = 57;

  while (1) {
    int f = 0;
    for (int i = 24; i >= 0 && !f; i--) {
      if (n % p[i] == 0) {
        printf("%d ", p[i]);
        n /= p[i]; f = 1;
      }
    }
    if (!f) break;
  }

  printf("\n");
}
```

Quiz 6: Tribute to Grothendieck

What does this code print?

```
char* p = "\2\3\5\7\v\r\x11\x13\x17\x1d\x1f%)+/5;=CGIOSYa";

int main(void)
{
  int n = 57;

  while (1) {
    int f = 0;
    for (int i = 24; i >= 0 && !f; i--) {
      if (n % p[i] == 0) {
        printf("%d ", p[i]);
        n /= p[i]; f = 1;
      }
    }
    if (!f) break;
  }

  printf("\n");
}
```

1. 57 19 3 1 1 ... (Infinite loop)

Quiz 6: Tribute to Grothendieck

What does this code print?

```
char* p = "\2\3\5\7\v\r\x11\x13\x17\x1d\x1f%)+/5;=CGIOSYa";

int main(void)
{
  int n = 57;

  while (1) {
    int f = 0;
    for (int i = 24; i >= 0 && !f; i--) {
      if (n % p[i] == 0) {
        printf("%d ", p[i]);
        n /= p[i]; f = 1;
      }
    }
    if (!f) break;
  }

  printf("\n");
}
```

1. 57 19 3 1 1 ... (Infinite loop)
2. Divide by zero error

Quiz 6: Tribute to Grothendieck

What does this code print?

```
char* p = "\2\3\5\7\v\r\x11\x13\x17\x1d\x1f%)+/5;=CGIOSYa";

int main(void)
{
  int n = 57;

  while (1) {
    int f = 0;
    for (int i = 24; i >= 0 && !f; i--) {
      if (n % p[i] == 0) {
        printf("%d ", p[i]);
        n /= p[i]; f = 1;
      }
    }
    if (!f) break;
  }

  printf("\n");
}
```

1. 57 19 3 1 1 ... (Infinite loop)
2. Divide by zero error
3. 19 3

Quiz 6: Tribute to Grothendieck

What does this code print?

```
char* p = "\2\3\5\7\v\r\x11\x13\x17\x1d\x1f%)+/5;=CGIOSYa";

int main(void)
{
  int n = 57;

  while (1) {
    int f = 0;
    for (int i = 24; i >= 0 && !f; i--) {
      if (n % p[i] == 0) {
        printf("%d ", p[i]);
        n /= p[i]; f = 1;
      }
    }
    if (!f) break;
  }

  printf("\n");
}
```

1. 57 19 3 1 1 ... (Infinite loop)
2. Divide by zero error
3. 19 3
4. 57 1

Quiz 6: Tribute to Grothendieck

What does this code print?

```
char* p = "\2\3\5\7\v\r\x11\x13\x17\x1d\x1f%)+/5;=CGIOSYa";

int main(void)
{
  int n = 57;

  while (1) {
    int f = 0;
    for (int i = 24; i >= 0 && !f; i--) {
      if (n % p[i] == 0) {
        printf("%d ", p[i]);
        n /= p[i]; f = 1;
      }
    }
    if (!f) break;
  }

  printf("\n");
}
```

1. 57 19 3 1 1 ... (Infinite loop)
2. Divide by zero error
3. 19 3
4. 57 1

Answer: 19 3.

Quiz 6: Tribute to Grothendieck

What does this code print?

```
char* p = "\2\3\5\7\v\r\x11\x13\x17\x1d\x1f%)+/5;=CGIOSYa";

int main(void)
{
  int n = 57;

  while (1) {
    int f = 0;
    for (int i = 24; i >= 0 && !f; i--) {
      if (n % p[i] == 0) {
        printf("%d ", p[i]);
        n /= p[i]; f = 1;
      }
    }
    if (!f) break;
  }

  printf("\n");
}
```

1. 57 19 3 1 1 ... (Infinite loop)
2. Divide by zero error
3. 19 3
4. 57 1

Answer: 19 3. This is a really small prime factorisation algorithm.

Quiz 6: Tribute to Grothendieck

What does this code print?

```
char* p = "\2\3\5\7\v\r\x11\x13\x17\x1d\x1f%)+/5;=CGIOSYa";

int main(void)
{
  int n = 57;

  while (1) {
    int f = 0;
    for (int i = 24; i >= 0 && !f; i--) {
      if (n % p[i] == 0) {
        printf("%d ", p[i]);
        n /= p[i]; f = 1;
      }
    }
    if (!f) break;
  }

  printf("\n");
}
```

1. 57 19 3 1 1 ... (Infinite loop)
2. Divide by zero error
3. 19 3
4. 57 1

Answer: 19 3. This is a really small prime factorisation algorithm. The primes are embedded in the ‘string’ p as individual characters.

Quiz 6: Tribute to Grothendieck

What does this code print?

```
char* p = "\2\3\5\7\v\r\x11\x13\x17\x1d\x1f%)+/5;=CGIOSYa";

int main(void)
{
  int n = 57;

  while (1) {
    int f = 0;
    for (int i = 24; i >= 0 && !f; i--) {
      if (n % p[i] == 0) {
        printf("%d ", p[i]);
        n /= p[i]; f = 1;
      }
    }
    if (!f) break;
  }

  printf("\n");
}
```

1. 57 19 3 1 1 ... (Infinite loop)
2. Divide by zero error
3. 19 3
4. 57 1

Answer: 19 3. This is a really small prime factorisation algorithm. The primes are embedded in the 'string' p as individual characters. Thus it only works for sufficiently small numbers.

Weird Literals

Weird Literals

This is Python code which was used to generate the primes array from the previous slide.

Weird Literals

This is Python code which was used to generate the primes array from the previous slide.

```
#!/usr/bin/env python3

import re

c_bytes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
           53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

# Convert the bytes into a C string
c_string = ''
for byte in c_bytes:
    if byte < 8:
        # Lower bytes are particularly common, so use their single octal digit form
        c_string += f'\\{byte}'
    elif byte in range(7, 14):
        # These bytes have a specific single character representation
        c_string += ['\\a', '\\b', '\\t', '\\n', '\\v', '\\f', '\\r'][byte - 7]
    elif byte in range(32, 127):
        # Handle " and \ specially, but just use the literal character otherwise
        if byte == 34:
            c_string += '\\"'
        elif byte == 92:
            c_string += '\\\\'
        else:
            c_string += chr(byte)
```

Weird Literals

This is Python code which was used to generate the primes array from the previous slide.

```
    else:
        # For other characters, use the \xHH representation
        c_string += f'\\x{byte:02x}'

# Deal with a weird quirk in C which Nick will explain, and add double quotes
c_string = '"' + re.sub(r'(\\x[0-9a-f][0-9a-f])([0-9A-Fa-f])', r'\1""\2', c_string) + '"'

# Print the resulting C string
print(c_string)
```

Weird Literals

Weird Literals

This is especially good for dealing with graphics. We can revisit our DNA printer example from the quiz slide earlier.

Weird Literals

This is especially good for dealing with graphics. We can revisit our DNA printer example from the quiz slide earlier.

```
for (int i = 0, j = 0, k = 0; i < 200; j = (++i) / 10 % 10, k = i % 10) {
  printf("%c", " \\/~"[
    (k == j)
    + 2 * (k == 9 - j)
    + 3 * ((k > j && k < 9 - j) || (k < j && k > 9 - j))
  ]);

  if (k == 9)
    printf("\n");
}
```

Contents

What can we Remove?

What can we Remove?

- There are many reasons why you might want to reduce the size of your C code, even outside of obfuscation.

What can we Remove?

- There are many reasons why you might want to reduce the size of your C code, even outside of obfuscation.
- Not only does it make it harder to read, it makes it easier to fit into a small spot (like an Instagram bio — @sadsunshower by the way).

What can we Remove?

- There are many reasons why you might want to reduce the size of your C code, even outside of obfuscation.
- Not only does it make it harder to read, it makes it easier to fit into a small spot (like an Instagram bio — @sadsunshower by the way).
- It's also a separate kind of challenge — golfing / code golf.

What can we Remove?

- There are many reasons why you might want to reduce the size of your C code, even outside of obfuscation.
- Not only does it make it harder to read, it makes it easier to fit into a small spot (like an Instagram bio — @sadsunshower by the way).
- It's also a separate kind of challenge — golfing / code golf.
- These same techniques can be applied to languages where size *does* matter.

What can we Remove?

- There are many reasons why you might want to reduce the size of your C code, even outside of obfuscation.
- Not only does it make it harder to read, it makes it easier to fit into a small spot (like an Instagram bio — @sadsunshower by the way).
- It's also a separate kind of challenge — golfing / code golf.
- These same techniques can be applied to languages where size *does* matter.
- JavaScript is usually minified in production websites

What can we Remove?

- There are many reasons why you might want to reduce the size of your C code, even outside of obfuscation.
- Not only does it make it harder to read, it makes it easier to fit into a small spot (like an Instagram bio — @sadsunshower by the way).
- It's also a separate kind of challenge — golfing / code golf.
- These same techniques can be applied to languages where size *does* matter.
- JavaScript is usually minified in production websites — since it's interpreted and sent over the web every byte of source code counts.

What can we Remove?

What can we Remove?

C fortunately doesn't require much whitespace as part of its syntax.

What can we Remove?

C fortunately doesn't require much whitespace as part of its syntax. There are two main places where whitespace matters:

What can we Remove?

C fortunately doesn't require much whitespace as part of its syntax. There are two main places where whitespace matters:

- Pre-processor macros like `#define` and `#include` must be on their own line.

What can we Remove?

C fortunately doesn't require much whitespace as part of its syntax. There are two main places where whitespace matters:

- Pre-processor macros like `#define` and `#include` must be on their own line.
- There must be a space between a variable/function and its (return) type: e.g. `int foo` or `void bar()`.

What can we Remove?

C fortunately doesn't require much whitespace as part of its syntax. There are two main places where whitespace matters:

- Pre-processor macros like `#define` and `#include` must be on their own line.
- There must be a space between a variable/function and its (return) type: e.g. `int foo` or `void bar()`.
- Note that this *doesn't* apply if the variable is a pointer, `int*foo` is acceptable and parses fine.

What can we Remove?

C fortunately doesn't require much whitespace as part of its syntax. There are two main places where whitespace matters:

- Pre-processor macros like `#define` and `#include` must be on their own line.
- There must be a space between a variable/function and its (return) type: e.g. `int foo` or `void bar()`.
- Note that this *doesn't* apply if the variable is a pointer, `int*foo` is acceptable and parses fine.

Additionally, there are a number of places where you specifically can't *add* whitespace, like in the middle of an identifier.

What can we Remove?

C fortunately doesn't require much whitespace as part of its syntax. There are two main places where whitespace matters:

- Pre-processor macros like `#define` and `#include` must be on their own line.
- There must be a space between a variable/function and its (return) type: e.g. `int foo` or `void bar()`.
- Note that this *doesn't* apply if the variable is a pointer, `int*foo` is acceptable and parses fine.

Additionally, there are a number of places where you specifically can't *add* whitespace, like in the middle of an identifier.
This is important if you want to shape your code in a particular way.

What can we Remove?

What can we Remove?

As discussed before, variables should be reduced to single-character names wherever possible.

What can we Remove?

As discussed before, variables should be reduced to single-character names wherever possible.
It's unlikely that you'll ever have more than 52 variables in the same scope (`a-z` and `A-Z`),

What can we Remove?

As discussed before, variables should be reduced to single-character names wherever possible.
It’s unlikely that you’ll ever have more than 52 variables in the same scope (`a-z` and `A-Z`), `_` is a valid identifier in case of emergency.

What can we Remove?

As discussed before, variables should be reduced to single-character names wherever possible.

It's unlikely that you'll ever have more than 52 variables in the same scope (`a-z` and `A-Z`), `_` is a valid identifier in case of emergency.

Also as discussed before, you should try to reuse variables as much as possible.

What can we Remove?

As discussed before, variables should be reduced to single-character names wherever possible.

It's unlikely that you'll ever have more than 52 variables in the same scope (`a-z` and `A-Z`), `_` is a valid identifier in case of emergency.

Also as discussed before, you should try to reuse variables as much as possible.

Not only does this make it less likely you'll need a two-letter variable name, it also avoids extra declarations.

What can we Remove?

What can we Remove?

C has an odd quirk where if a variable type is left out, it should be assumed to be an integer. The following programs are equivalent:

What can we Remove?

C has an odd quirk where if a variable type is left out, it should be assumed to be an integer. The following programs are equivalent:

```c
int fib(int n) {
  int a = 0;
  int b = 1;

  for (int i = 0; i < n - 1; i++) {
    int tmp = a + b;
    a = b;
    b = tmp;
  }

  return b;
}

int main(void) {
  for (int i = 1; i < 20; i++) {
    printf("F_%d = %d\n", i, fib(i));
  }
}
```

What can we Remove?

C has an odd quirk where if a variable type is left out, it should be assumed to be an integer. The following programs are equivalent:

```c
int fib(int n) {
  int a = 0;
  int b = 1;

  for (int i = 0; i < n - 1; i++) {
    int tmp = a + b;
    a = b;
    b = tmp;
  }

  return b;
}

int main(void) {
  for (int i = 1; i < 20; i++) {
    printf("F_%d = %d\n", i, fib(i));
  }
}
```

```c
a, b, i, j, n, tmp;

fib(n) {
  a = 0;
  b = 1;

  for (i = 0; i < n - 1; i++) {
    tmp = a + b;
    a = b;
    b = tmp;
  }

  return b;
}

main(void) {
  for (j = 1; j < 20; j++) {
    printf("F_%d = %d\n", j, fib(j));
  }
}
```

Most C compilers will complain heavily when you try this, but it saves 4 characters from every `int` you need to declare, and at least 4 characters from every program (due to `main`).

What can we Remove?

What can we Remove?

- These measures are not likely to be effective on their own for golfing, although they do help with obfuscation.

What can we Remove?

- These measures are not likely to be effective on their own for golfing, although they do help with obfuscation.
- For effective golfing, you really need large structural changes to your code.

What can we Remove?

- These measures are not likely to be effective on their own for golfing, although they do help with obfuscation.
- For effective golfing, you really need large structural changes to your code.
- For example, replacing all the varialbe names is not as effective as finding a way to remove two whole loops.

Terminal Graphics

Terminal Graphics

- *American National Standards Institute* (ANSI) standardised some 'control sequences' for computer terminals.

Terminal Graphics

- *American National Standards Institute* (ANSI) standardised some 'control sequences' for computer terminals.
- These allow terminals to do things such as display colour and graphics with just text.

Terminal Graphics

- *American National Standards Institute* (ANSI) standardised some 'control sequences' for computer terminals.
- These allow terminals to do things such as display colour and graphics with just text.
- We can use these to do very cheap and quick graphics with just `printf`. No graphics library or `libncurses` required!

Terminal Graphics

Terminal Graphics

- `\e[n;mH` — Moves the cursor to row n, column n.

Terminal Graphics

- `\e[n;mH` — Moves the cursor to row n, column n.
- `\e[3J` — Clear screen

Terminal Graphics

- `\e[n;mH` — Moves the cursor to row n, column n.
- `\e[3J` — Clear screen
- `\e[nm` — Sets colour to `n`. Colour references are available online.

Terminal Graphics

- `\e[n;mH` — Moves the cursor to row n, column n.
- `\e[3J` — Clear screen
- `\e[nm` — Sets colour to `n`. Colour references are available online.
- `https://misc.flogisoft.com/bash/tip_colors_and_formatting`

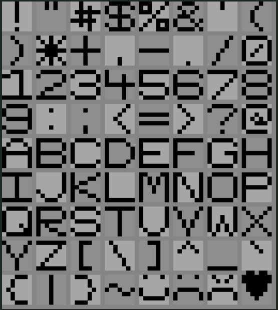
Fonts on a Budget

Fonts on a Budget

Bitmapped fonts are very simple and surprisingly readable. They also can be used naturally for images and terminal graphics.
More importantly for us, they don't take up much space.

Fonts on a Budget

Bitmapped fonts are very simple and surprisingly readable. They also can be used naturally for images and terminal graphics.
More importantly for us, they don't take up much space.
Introducing *Workshop Mono*, our very own 7x7 pixel bitmap font.



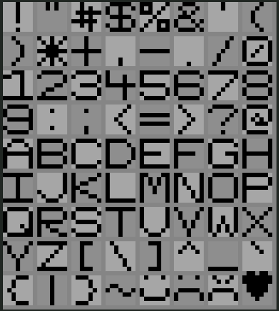
Fonts on a Budget

Bitmapped fonts are very simple and surprisingly readable. They also can be used naturally for images and terminal graphics.

More importantly for us, they don't take up much space.

Introducing *Workshop Mono*, our very own 7x7 pixel bitmap font.

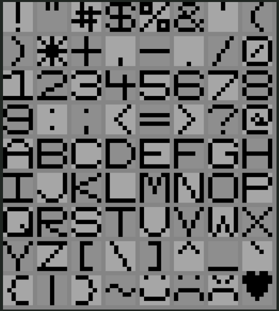
- It only supports upper case letters. Characters are drawn in ASCII order and are actually 8x8 pixels with 1 pixel of padding on the bottom and right. We also have some extra fun characters.



Fonts on a Budget

Bitmapped fonts are very simple and surprisingly readable. They also can be used naturally for images and terminal graphics.
More importantly for us, they don't take up much space.
Introducing *Workshop Mono*, our very own 7x7 pixel bitmap font.

- It only supports upper case letters. Characters are drawn in ASCII order and are actually 8x8 pixels with 1 pixel of padding on the bottom and right. We also have some extra fun characters.
- 8x8 gives us an advantage, we can directly embed the font into our C program. Each character will take up 8 bytes, or about 1-4 characters in the source code.



Fonts on a Budget

Fonts on a Budget

This is our strategy for embedding the fonts in C:

- Go through each 8x8 character, and for each of those characters read one 8 pixel column at a time.
- Convert that column into an 8-bit number; 1 = filled in pixel, 0 = blank pixel.
- Append this 8-bit number to a C string as one character / byte.

Not all characters can be represented by ASCII cleanly, so we'll need to use special conversion to represent them in the C string.

Fonts on a Budget

Fonts on a Budget

```
#!/usr/bin/env python3
import PIL.Image, re

font_bytes = []

# Open the image
with PIL.Image.open('workshop-mono.png') as image:
    # Scan each character, going along each row left to right
    for row in range(0, image.height, 8):
        for col in range(0, image.width, 8):
            # Append each vertical line as a new byte
            for line in range(8):
                bits = 0
                # Going "backwards" is a neat compression trick which Nick will explain
                for pixel in range(7, -1, -1):
                    # Black = 1, white = 0
                    # [:3] ignores any alpha channel if it exists
                    if image.getpixel((col + line, row + pixel))[:3] == (0, 0, 0):
                        bits = (bits << 1) | 1
                    else:
                        bits = (bits << 1)

                # Write this byte to the list of bytes
                font_bytes.append(bits)
```

This is our script for converting bitmap font images to C strings.

Fonts on a Budget

Fonts on a Budget

```
    # Convert the font bytes into a C string
    c_string = ''
    for byte in font_bytes:
        if byte < 8:
            # Lower bytes are particularly common, so use their single octal digit form
            c_string += f'\\{byte}'
        elif byte in range(7, 14):
            # These bytes have a specific single character representation
            c_string += ['\\a', '\\b', '\\t', '\\n', '\\f', '\\r'][byte - 7]
        elif byte in range(32, 127):
            # Handle " and \ specially, but just use the literal character otherwise
            if byte == 34:
                c_string += '\\"'
            elif byte == 92:
                c_string += '\\\\'
            else:
                c_string += chr(byte)
        else:
            # For other characters, use the \xHH representation
            c_string += f'\\x{byte:02x}'
```

This is our script for converting bitmap font images to C strings.

Fonts on a Budget

Fonts on a Budget

```
# Deal with a weird quirk in C which Nick will explain, and add double quotes
c_string = '"' + re.sub(r'(\\x[0-9a-f][0-9a-f])([0-9A-Fa-f])', r'\1""\2', c_string) + '"'

# Print the resulting C string
print(c_string)
```

This is our script for converting bitmap font images to C strings.

Fonts on a Budget

Fonts on a Budget

Using this script we'll notice a problem:

```
$ ./bitmap-fonts.py | wc -c
1073
```

We have way too many characters. There are a number of ways we can deal with this:

- Each character has at least one column of spacing which will appear as '`\0`', so we can skip this column entirely when encoding.

Fonts on a Budget

Using this script we'll notice a problem:

```
$ ./bitmap-fonts.py | wc -c
1073
```

We have way too many characters. There are a number of ways we can deal with this:

- Each character has at least one column of spacing which will appear as '`\0`', so we can skip this column entirely when encoding. But we need to remember to add spacing between characters.

Fonts on a Budget

Using this script we'll notice a problem:

```
$ ./bitmap-fonts.py | wc -c
1073
```

We have way too many characters. There are a number of ways we can deal with this:

- Each character has at least one column of spacing which will appear as '`\0`', so we can skip this column entirely when encoding. But we need to remember to add spacing between characters.
- There are many other columns which are entirely blank which will also appear as '`\0`'. We could replace '`\0`' with another single character that doesn't appear in the C string (e.g. 'w'), halving the size of each of these zeroes when embedded in the C string.

Fonts on a Budget

Using this script we'll notice a problem:

```
$ ./bitmap-fonts.py | wc -c
1073
```

We have way too many characters. There are a number of ways we can deal with this:

- Each character has at least one column of spacing which will appear as '`\0`', so we can skip this column entirely when encoding. But we need to remember to add spacing between characters.
- There are many other columns which are entirely blank which will also appear as '`\0`'. We could replace '`\0`' with another single character that doesn't appear in the C string (e.g. 'w'), halving the size of each of these zeroes when embedded in the C string. However, we need to remember to replace them with zeroes inside the code.

Fonts on a Budget

Using this script we'll notice a problem:

```
$ ./bitmap-fonts.py | wc -c
1073
```

We have way too many characters. There are a number of ways we can deal with this:

- Each character has at least one column of spacing which will appear as '`\0`', so we can skip this column entirely when encoding. But we need to remember to add spacing between characters.
- There are many other columns which are entirely blank which will also appear as '`\0`'. We could replace '`\0`' with another single character that doesn't appear in the C string (e.g. 'w'), halving the size of each of these zeroes when embedded in the C string. However, we need to remember to replace them with zeroes inside the code.
- We could use a smaller font size such as 5x5.

Fonts on a Budget

Using this script we'll notice a problem:

```
$ ./bitmap-fonts.py | wc -c
1073
```

We have way too many characters. There are a number of ways we can deal with this:

- Each character has at least one column of spacing which will appear as '\0', so we can skip this column entirely when encoding. But we need to remember to add spacing between characters.
- There are many other columns which are entirely blank which will also appear as '\0'. We could replace '\0' with another single character that doesn't appear in the C string (e.g. 'w'), halving the size of each of these zeroes when embedded in the C string. However, we need to remember to replace them with zeroes inside the code.
- We could use a smaller font size such as 5x5. But it would be harder to decode, since individual characters wouldn't fit neatly into bytes.

Fonts on a Budget

Using this script we'll notice a problem:

```
$ ./bitmap-fonts.py | wc -c
1073
```

We have way too many characters. There are a number of ways we can deal with this:

- Each character has at least one column of spacing which will appear as '`\0`', so we can skip this column entirely when encoding. But we need to remember to add spacing between characters.
- There are many other columns which are entirely blank which will also appear as '`\0`'. We could replace '`\0`' with another single character that doesn't appear in the C string (e.g. 'w'), halving the size of each of these zeroes when embedded in the C string. However, we need to remember to replace them with zeroes inside the code.
- We could use a smaller font size such as 5x5. But it would be harder to decode, since individual characters wouldn't fit neatly into bytes.
- We could use some form of run-length encoding.

Fonts on a Budget

Using this script we'll notice a problem:

```
$ ./bitmap-fonts.py | wc -c
1073
```

We have way too many characters. There are a number of ways we can deal with this:

- Each character has at least one column of spacing which will appear as '`\0`', so we can skip this column entirely when encoding. But we need to remember to add spacing between characters.
- There are many other columns which are entirely blank which will also appear as '`\0`'. We could replace '`\0`' with another single character that doesn't appear in the C string (e.g. 'w'), halving the size of each of these zeroes when embedded in the C string. However, we need to remember to replace them with zeroes inside the code.
- We could use a smaller font size such as 5x5. But it would be harder to decode, since individual characters wouldn't fit neatly into bytes.
- We could use some form of run-length encoding. But it could make the C string actually larger depending on the way it's used, and would make decoding more complicated.

Fonts on a Budget

Fonts on a Budget

This is a very tricky engineering problem. We need to consider how many characters we’ll save by compressing various parts of the font vs. how many we’ll gain with the compression code.

Fonts on a Budget

This is a very tricky engineering problem. We need to consider how many characters we'll save by compressing various parts of the font vs. how many we'll gain with the compression code.
The first two options seem to be the most viable. We can even repeat the second option for other characters for more savings. We now have a much more reasonable count:

Fonts on a Budget

This is a very tricky engineering problem. We need to consider how many characters we'll save by compressing various parts of the font vs. how many we'll gain with the compression code.

The first two options seem to be the most viable. We can even repeat the second option for other characters for more savings. We now have a much more reasonable count:

```
$ ./bitmap-fonts.py | wc -c
807
```

Fonts on a Budget

This is a very tricky engineering problem. We need to consider how many characters we'll save by compressing various parts of the font vs. how many we'll gain with the compression code.

The first two options seem to be the most viable. We can even repeat the second option for other characters for more savings. We now have a much more reasonable count:

```
$ ./bitmap-fonts.py | wc -c
807
```

Now we can embed this font into our C code and start to use it.

Contents

Worked Examples

Worked Examples

We will now take some time to pick apart some finished examples.

Worked Examples

We will now take some time to pick apart some finished examples.

- My Discord status.

Worked Examples

We will now take some time to pick apart some finished examples.

- My Discord status.
- My Instagram bio.

Worked Examples

We will now take some time to pick apart some finished examples.

- My Discord status.
- My Instagram bio.
- The event banner code.