Strange things can, and will, happen if you break the rules of the language. In C there is a very detailed contract between the programmer and the compiler that you need to understand well. It is sometimes said that upon encountering a contract violation in C, it is legal for the compiler to make nasal demons fly out of your nose. In practice, however, compilers usually do not try to pull pranks on you, but even when trying to do their best they might give you big surprises.

In this talk we will study actual machine code generated by snippets of both legal and illegal C code. We will use these examples while studying and discussing parts of the ISO/IEC 9899 standard (the C standard).

A 90 minute session at ACCU 2013 in Bristol UK
Thursday, April 11, 2013
Warning: this presentation contains a lot of crap code. Don’t try this at home.
Warning: this presentation contains a lot of crap code. Don’t try this at home.
Warning: this presentation contains a lot of crap code. Don’t try this at work.
Exercise

This code is **undefined behavior**, but what do you think actually happens if you compile, link and run it in your development environment?

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```
Exercise

This code is **undefined behavior**, but what do you think actually happens if you compile, link and run it in your development environment?

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

foo.c

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):
Exercise

This code is **undefined behavior**, but what do you think actually happens if you compile, link and run it in your development environment?

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc foo.c && .a.out
```

Exercise

This code is **undefined behavior**, but what do you think actually happens if you compile, link and run it in your development environment?

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc foo.c && ./a.out
12
```
Exercise

This code is **undefined behavior**, but what do you think actually happens if you compile, link and run it in your development environment?

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d
", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc foo.c && ./a.out
12
$ clang foo.c && ./a.out
```
Exercise

This code is **undefined behavior**, but what do you think actually happens if you compile, link and run it in your development environment?

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc foo.c && ./a.out
12
$ clang foo.c && ./a.out
11
```
Exercise

This code is **undefined behavior**, but what do you think actually happens if you compile, link and run it in your development environment?

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0, 2, 4, 6, 8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc foo.c && ./a.out
12
$ clang foo.c && ./a.out
11
$ icc foo.c && ./a.out
```
Exercise

This code is **undefined behavior**, but what do you think actually happens if you compile, link and run it in your development environment?

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0, 2, 4, 6, 8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc foo.c && ./a.out
12
$ clang foo.c && ./a.out
11
$ icc foo.c && ./a.out
13
```
```c
#include <stdio.h>

int main(void)
{
    int v[] = {0, 2, 4, 6, 8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```
It is always a good idea to add some flags for better diagnostics.
On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

It is always a good idea to add some flags for better diagnostics.
It is always a good idea to add some flags for better diagnostics.

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
```
```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):
```
$ gcc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
```

12

It is always a good idea to add some flags for better diagnostics.
```c
#include <stdio.h>

int main(void)
{
    int v[] = {0, 2, 4, 6, 8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

It is always a good idea to add some flags for better diagnostics.
#include <stdio.h>

int main(void)
{
    int v[] = {0, 2, 4, 6, 8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

$ gcc -std=c99 -0 -Wall -Wextra -pedantic foo.c && ./a.out
12
$ clang -std=c99 -0 -Wall -Wextra -pedantic foo.c && ./a.out
11

It is always a good idea to add some flags for better diagnostics.
#include <stdio.h>

int main(void)
{
    int v[] = {0, 2, 4, 6, 8};
    int i = 1;
    int n = i + v[+i] + v[+i];
    printf("%d\n", n);
}

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

$ gcc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
12
$ clang -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
11
$ icc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out

It is always a good idea to add some flags for better diagnostics.
It is always a good idea to add some flags for better diagnostics.
```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```
Can we learn anything about C by studying code like this?

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```
Can we learn anything about C by studying code like this?

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

YES!
Can we learn anything about C by studying code like this?

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

If your answer is a definitive NO, then I am afraid that the rest of this presentation is probably a waste of your time.
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
First let us agree on the theoretical part first.
First let us agree on the theoretical part first.

The code is undefined behavior because it tries to modify a value twice between two sequence points. The rules of sequencing has been violated. The standard says:
First let us agree on the theoretical part first.

The code is undefined behavior because it tries to modify a value twice between two sequence points. The rules of sequencing has been violated. The standard says:

[6.5 Expressions]

... Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.

The grouping of operators and operands is indicated by the syntax. Except as specified later (for the function-call (), &&, ||, ?:, and comma operators), the order of evaluation of subexpressions and the order in which side effects take place are both unspecified.

...
First let us agree on the theoretical part first.

The code is undefined behavior because it tries to modify a value twice between two sequence points. The rules of sequencing has been violated. The standard says:

6.5 Expressions

Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.

The grouping of operators and operands is indicated by the syntax. Except as specified later (for the function-call (), &&, ||, ?:, and comma operators), the order of evaluation of subexpressions and the order in which side effects take place are both unspecified.

It takes a while to really understand those words. However, the key thing is the last sentence, which is basically saying: **the order of evaluation is mostly unspecified.** The rest is just a necessary consequence of this particular “language feature”.

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```
#include <stdio.h>

int main(void) {
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}

First let us agree on the theoretical part first.

The code is undefined behavior because it tries to modify a value twice between two sequence points. The rules of sequencing has been violated. The standard says:

[6.5 Expressions]

... Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.

The grouping of operators and operands is indicated by the syntax. Except as specified later (for the function-call (), &&, ||, ?:, and comma operators), the order of evaluation of subexpressions and the order in which side effects take place are both unspecified.

... It takes a while to really understand those words. However, the key thing is the last sentence, which is basically saying: **the order of evaluation is mostly unspecified**. The rest is just a necessary consequence of this particular “language feature”.

Since evaluation order here is unspecified, the expression does not make sense, and therefore the standard just says that this is **undefined behavior**.
```java
public class Foo {
    public static void main(String args[]) {
        int v[] = {0,2,4,6,8};
        int i = 1;
        int n = i + v[++i] + v[++i];
        System.out.println(n);
    }
}
```

In Java, the calculated value is guaranteed to be 11.

```bash
$ javac Foo.java && java Foo
11
```
```java
public class Foo {
    public static void main(String args[]) {
        int v[] = {0,2,4,6,8};
        int i = 1;
        int n = i + v[++i] + v[++i];
        System.out.println(n);
    }
}
```

“Mostly unspecified evaluation order” is a rather unique feature of C (and C++ and Fortran). Most other languages guarantees a certain evaluation order.

```c
#include <stdio.h>

int main(void) {
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

In Java, the calculated value is guaranteed to be 11.

```
$ javac Foo.java && java Foo
11
```
#include <stdio.h>

int main(void)
{
    int v[] = {0, 2, 4, 6, 8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
OK, enough theory. Let's take a look under the hood and try to find out what different C compilers are actually doing here.
Let’s first change the code slightly, so it is easier to see what is happening.
Let's first change the code slightly, so it is easier to see what is happening.

```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```


```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

Let's first change the code slightly, so it is easier to see what is happening.

The undefined behavior is still the same.
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}

Let's first change the code slightly, so it is easier to see what is happening.

The undefined behavior is still the same.

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

$ /usr/bin/gcc main.c && ./a.out
12
$ /usr/bin/clang main.c && ./a.out
11
$ /usr/bin/icc main.c && ./a.out
13
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

Let's first change the code slightly, so it is easier to see what is happening.

The undefined behavior is still the same.

Also, to make the assembler easier to read we target 32-bit architecture, tune for i386, and make sure to turn off all optimization. The observed behavior is still the same...
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ /usr/bin/gcc -O0 -std=c99 -m32 -mtune=i386 main.c && ./a.out
12
$ /usr/bin/clang -O0 -std=c99 -m32 -mtune=i386 main.c && ./a.out
11
$ /usr/bin/icc -O0 -std=c99 -m32 -mtune=i386 main.c && ./a.out
13
```

Let's first change the code slightly, so it is easier to see what is happening.

The undefined behavior is still the same.

Also, to make the assembler easier to read we target 32-bit architecture, tune for i386, and make sure to turn off all optimization. The observed behavior is still the same...
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

I used gdb to disassemble the output, eg

```
$ /usr/bin/gcc -O0 -std=c99 -m32 -mtune=i386 main.c
$ ./a.out
```
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

```
$ /usr/bin/gcc -O0 -std=c99 -m32 -mtune=i386 main.c
$ ./a.out
12
```
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

I used gdb to disassemble the output, eg

```
$ /usr/bin/gcc -O0 -std=c99 -m32 -mtune=i386 main.c
$ ./a.out
12
$ gdb ./a.out
```
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}

I used gdb to disassemble the output, eg:

$ /usr/bin/gcc -O0 -std=c99 -m32 -mtune=i386 main.c
$ ./a.out
12
$ gdb ./a.out
(gdb) set disassembly-flavor intel
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

This is the standard preamble for the function. Here it allocates space on the stack for variables with automatic storage duration (local variables).
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>
int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

Here a[0] is loaded into a register, 1 is added and the value is written into the memory location dedicated for the local variable i. The variable i has now been initialized.
Here a[0] is loaded into a register, I is added and the value is written into the memory location dedicated for the local variable i. The variable i has now been initialized.

Now the interesting stuff starts!
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + (a[++i] + b[++i]);
    printf("%d\n", n);
}

GCC
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

`gcc` first wants to evaluate subexpression `a[++i]`. The value stored in `i` is loaded into a register, then increased by 1, and the value is stored back to memory. Then `i` is loaded into register again and used to calculate the index into array `a` and that particular value is loaded into register. The subexpression is evaluated, and the value of this subexpression should now be 4. The stored value of `i` is 2.
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

Now, i is loaded into register, and added with the newly calculated value of a[++] + b[++] is 6. The result of subexpression i + a[++] is stored in ecx and should now be 6.
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

Now it is time to evaluate subexpression \(b[+i]\). Load \(i\) into register, increase by one, store value to memory. Value of \(i\) is now 3. Load \(i\) into register again, use to calculate index into array \(b\), and load that value into register. The value is 6.
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}

The newly calculated subexpression b[++] is now stored in edx. The old subexpression i + a[++] is stored in ecx. They are added together.
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

The newly calculated subexpression `b[++i]` is now stored in edx. The old subexpression `i + a[++i]` is stored in ecx. They are added together.

6 + 6 is 12
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

The newly calculated subexpression \(b[++i]\) is now stored in \(edx\). The old subexpression \(i + a[++i]\) is stored in \(ecx\). They are added together.

6 + 6 is 12

and the result is stored into the memory location allocated for local variable \(n\).
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

The value of `n` is printed.
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

The value of `n` is printed: 12
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

This is just the postamble of the main function. Store 0 as the return value, load eax with the return value. Pop the stack and return to the caller.
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

This is the interesting part of the code.
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

text
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

This is the interesting part of the code.

Surprised?
Most programmers intuitively think that an expression ought to be evaluated from left to right. Eg like this:
Most programmers intuitively think that an expression ought to be evaluated from left to right. Eg like this:

```
i + a[++i] + b[++i]
1 + a[++i] + b[++i]
1 + a[1+] + b[++i]
1 + a[2] + b[++i]
1 + 4 + b[1+]
5 + b[+]
5 + b[+2]
5 + b[3]
5 + 6
11
```
Most programmers intuitively think that an expression ought to be evaluated from left to right. Eg like this:

```
i + a[++i] + b[++i]
1 + a[++i] + b[++i]
1 + a[++1] + b[++i]
1 + a[2] + b[++i]
1 + 4 + b[++i]
5 + b[++i]
5 + b[++2]
5 + b[3]
5 + 6
11
```

This is guaranteed to happen in most other languages, but not in C, and as we have seen, certainly not with GCC. The evaluation order of this expression is unspecified, and we are not allowed to assume anything about the side effects that happens when evaluating subexpressions...
Most programmers intuitively think that an expression ought to be evaluated from left to right. Eg like this:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluation Order</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i + a[++i] + b[++i]</code></td>
<td><code>1 + a[++1] + b[++i]</code></td>
</tr>
<tr>
<td><code>1 + a[++i] + b[++i]</code></td>
<td><code>1 + a[2] + b[++i]</code></td>
</tr>
<tr>
<td><code>1 + 4 + b[++i]</code></td>
<td><code>1 + 4 + b[++i]</code></td>
</tr>
<tr>
<td><code>5 + b[++i]</code></td>
<td><code>5 + b[++2]</code></td>
</tr>
<tr>
<td><code>5 + b[3]</code></td>
<td><code>5 + 6</code></td>
</tr>
<tr>
<td><code>5 + 6</code></td>
<td><code>11</code></td>
</tr>
</tbody>
</table>

This is guaranteed to happen in most other languages, but not in C, and as we have seen, certainly not with GCC. The evaluation order of this expression is unspecified, and we are not allowed to assume anything about the sideeffects that happens when evaluating subexpressions...

The evaluation we just observed looked like this:
Most programmers intuitively think that an expression ought to be evaluated from left to right. Eg like this:

```plaintext
i + a[++i] + b[++i]
i + a[++i] + b[++i]
i + a[2] + b[++i]
1 + 4 + b[++i]
5 + b[++i]
5 + b[++2]
5 + b[3]
5 + 6
11
```

This is guaranteed to happen in most other languages, but not in C, and as we have seen, certainly not with GCC. The evaluation order of this expression is unspecified, and we are not allowed to assume anything about the sideeffects that happens when evaluating subexpressions...

The evaluation we just observed looked like this:

```plaintext
i + a[++i] + b[++i]
i + a[++1] + b[++i]
i + a[2] + b[++i]
i + 4 + b[++i]
2 + 4 + b[++i]
6 + b[++i]
6 + b[++2]
6 + b[3]
6 + 6
12
```
Most programmers intuitively think that an expression ought to be evaluated from left to right. Eg like this:

This is guaranteed to happen in most other languages, but not in C, and as we have seen, certainly not with GCC. The evaluation order of this expression is unspecified, and we are not allowed to assume anything about the sideeffects that happens when evaluating subexpressions...

The evaluation we just observed looked like this:

Is gcc doing anything wrong here? Not at all. The C standard explicitly says that the evaluation order here is unspecified, and therefore gcc can do whatever it wants. The C standard also says that this is undefined behavior, so gcc could have just skipped the calculation and said value of n is 42. And still be right.
push ebp
mov ebp, esp
sub esp, 0x18
call 0x1f1b <main+11>
pop eax
mov ecx, DWORD PTR [eax+0x115]
add ecx, 0x1
mov DWORD PTR [ebp-0xc], ecx
mov ecx, DWORD PTR [ebp-0xc]
add ecx, 0x1
mov DWORD PTR [ebp-0xc], ecx
mov ecx, DWORD PTR [ebp-0xc]
add ecx, 0x1
mov DWORD PTR [ebp-0xc], ecx
mov edx, DWORD PTR [ebp-0xc]
add ecx, edx
mov edx, DWORD PTR [ebp-0xc]
add edx, 0x1
mov DWORD PTR [ebp-0xc], edx
mov edx, DWORD PTR [ebp-0xc]
add edx, 0x1
mov DWORD PTR [ebp-0xc], edx
mov edx, DWORD PTR [ebp-0xc]
add edx, 0x1
mov DWORD PTR [ebp-0xc], edx
mov ecx, DWORD PTR [ebp-0x10]
mov edx, esp
mov DWORD PTR [edx+0x4], ecx
lea eax, [eax+0x95]
mov DWORD PTR [edx], eax
call 0x1f88 <dyld_stub_printf>
mov DWORD PTR [ebp-0x8], 0x0
mov eax, DWORD PTR [ebp-0x8]
mov DWORD PTR [ebp-0x4], eax
mov eax, DWORD PTR [ebp-0x4]
add esp, 0x18
pop ebp
ret
Here is a simple technique for writing executable pseudo-assembler.
Here is a simple technique for writing executable pseudo-assembler.

```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

struct reg {
    int a,b,c,d;
} reg;

int main(void) {
    int i, n;

    reg.c = a[0];
    reg.c += 1;
    i = reg.c;

    reg.c = i;
    reg.c += 1;
    i = reg.c;

    reg.c = a[reg.c];
    reg.d = i;
    reg.c += reg.d;

    reg.d = i;
    reg.d += 1;
    i = reg.d;
    reg.d = b[reg.d];

    reg.c += reg.d;
    n = reg.c;

    printf("%d\n", n);
}
```
Here is a simple technique for writing executable psudo-assembler.

```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

struct reg {
    int a,b,c,d;
} reg;

int main(void) {
    int i, n;

    reg.c = a[0];
    reg.c += 1;
    i = reg.c;

    reg.c = i;
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c = a[reg.c];

    reg.d = i;
    reg.c += reg.d;

    reg.d = i;
    reg.d += 1;
    i = reg.d;
    reg.d = i;
    reg.d = b[reg.d];

    reg.c += reg.d;
    n = reg.c;
    printf("%d\n", n);
}
```
Here is a simple technique for writing executable pseudo-assembler. The trick is to introduce some fake registers, and then spell out the assembler in pure C.
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

struct reg {
    int a, b, c, d;
} reg;

int main(void) {
    int i, n;

    reg.c = a[0];
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c += 1;
    i = reg.c;
    reg.c = i;

    reg.c = a[reg.c];
    reg.d = i;
    reg.c += reg.d;

    reg.d = i;
    reg.d += 1;
    i = reg.d;
    reg.d = i;
    reg.d = b[reg.d];

    reg.c += reg.d;
    n = reg.c;
    printf("%d\n", n);
}
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

struct reg {
    int a, b, c, d;
} reg;

int main(void) {
    int i, n;

    reg.c = a[0];
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c = a[reg.c];

    reg.d = i;
    reg.c += reg.d;
    reg.d = i;
    reg.d += 1;
    i = reg.d;
    reg.d = i;
    reg.d = b[reg.d];

    reg.c += reg.d;
    n = reg.c;

    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

struct reg {
    int a,b,c,d;
} reg;

int main(void) {
    int i, n;

    reg.c = a[0];
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c += 1;
    i = reg.c;
    reg.c += reg.d;
    reg.d = i;
    reg.c += reg.d;
    reg.d = b[reg.d];

    reg.c += reg.d;
    n = reg.c;
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

struct reg {
    int a,b,c,d;
} reg;

int main(void) {
    int i, n;
    reg.c = a[0];
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c = a[reg.c];
    reg.d = i;
    reg.c += reg.d;
    reg.d = i;
    reg.d += 1;
    i = reg.d;
    reg.d = i;
    reg.d = b[reg.d];
    reg.c += reg.d;
    n = reg.c;
    printf("%d
", n);
}
```
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

struct reg {
    int a, b, c, d;
} reg;

int main(void) {
    int i, n;

    reg.c = a[0];
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c = a[reg.c];
    reg.d = i;
    reg.c += reg.d;

    reg.d = i;
    reg.d += 1;
    i = reg.d;
    reg.d = i;
    reg.d = b[reg.d];

    reg.c += reg.d;
    n = reg.c;

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

#include <stdio.h>
int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

struct reg {
    int a,b,c,d;
} reg;

int main(void) {
    int i, n;
    reg.c = a[0];
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c = a[reg.c];
    reg.d = i;
    reg.c += reg.d;
    reg.d = i;
    reg.d += 1;
    i = reg.d;
    reg.d = i;
    reg.d = b[reg.d];
    reg.c += reg.d;
    n = reg.c;
    printf("%d\n", n);
}
gcc

```
push ebp
mov ebp, esp
sub esp, 0x18
call 0x1f1b <main+11>
pop eax
mov ecx, DWORD PTR [eax+0x115]
add ecx, 0x1
mov DWORD PTR [ebp-0xc], ecx
mov ecx, DWORD PTR [ebp-0xc]
add ecx, 0x1
mov DWORD PTR [ebp-0xc], ecx
mov ecx, DWORD PTR [eax+ecx*4+0x115]
mov edx, DWORD PTR [ebp-0xc]
add ecx, edx
mov edx, DWORD PTR [ebp-0xc]
add edx, 0x1
mov DWORD PTR [ebp-0xc], edx
mov edx, DWORD PTR [eax+edx*4+0x135]
add ecx, edx
mov DWORD PTR [ebp-0xc], ecx
mov ecx, DWORD PTR [ebp-0xc]
add ecx, 0x1
mov DWORD PTR [ebp-0xc], ecx
mov edx, DWORD PTR [eax+0x4], ecx
lea eax, [eax+0x95]
mov DWORD PTR [eax], eax
call 0x1f88 <dyld_stub_printf>
mov DWORD PTR [ebp-0x8], 0
mov eax, DWORD PTR [ebp-0x8]
mov eax, DWORD PTR [ebp-0x8]
mov DWORD PTR [ebp-0x4], eax
mov eax, DWORD PTR [ebp-0x4]
add esp, 0x18
pop ebp
ret
```
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

struct reg {
    int a, b, c, d;
} reg;

int main(void) {
    int i, n;
    reg.c = a[0];
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c = a[reg.c];
    reg.d = i;
    reg.c += reg.d;
    reg.d = i;
    reg.d += 1;
    i = reg.d;
    reg.d = i;
    reg.d = b[reg.d];
    reg.c += reg.d;
    n = reg.c;
    printf("%d\n", n);
}
```c

#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

struct reg {
    int a,b,c,d;
} reg;

int main(void) {
    int i, n;

    reg.c = a[0];
    reg.c += 1;
    i = reg.c;

    reg.c = i;
    reg.c += 1;
    i = reg.c;

    reg.c = a[reg.c];
    reg.d = i;
    reg.c += reg.d;

    reg.d = i;
    reg.d += 1;
    i = reg.d;
    reg.d = i;
    reg.d = b[reg.d];

    reg.c += reg.d;
    n = reg.c;

    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

struct reg {
    int a, b, c, d;
} reg;

int main(void) {
    int i, n;

    reg.c = a[0];
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c = a[reg.c];
    reg.d = i;
    reg.c += reg.d;

    reg.d = i;
    reg.d += 1;
    i = reg.d;
    reg.d = i;
    reg.d = b[reg.d];

    n = reg.c;
    printf("n = (i + a[++i]) + (b[++i])
", n);
}
```
gcc

```
push   ebp
mov    ebp,esp
sub    esp,0x18
call   0x1f1b <main+11>
pop    eax
mov    ecx,DWORD PTR [eax+0x115]  ; reg.c = a[0]
add    ecx,0x1
mov    DWORD PTR [ebp-0xc],ecx
mov    ecx,DWORD PTR [ebp-0xc]     ; reg.c += 1;
i = reg.c;
add    ecx,ecx
mov    edx,DWORD PTR [ebp-0xc]     ; reg.c = i;
add    edx,0x1
mov    DWORD PTR [ebp-0xc],edx     ; reg.c += 1;
i = reg.c;
mov    edx,DWORD PTR [ebp-0xc]     ; reg.c = i;
add    edx,edx
mov    DWORD PTR [ebp-0xc],edx     ; reg.c = a[reg.c];
add    edx,eax                     ; reg.d = i;
lea    eax,[eax+0x95]
mov    DWORD PTR [edx],eax         ; reg.d += 1;
i = reg.d;
mov    DWORD PTR [edx+0x4],ecx    ; reg.d = i;
mov    DWORD PTR [edx+0x4],eax     ; reg.d = b[reg.d];
add    esp,0x18
mov    DWORD PTR [ebp-0x8],0x0
mov    eax,DWORD PTR [ebp-0x8]
mov    DWORD PTR [ebp-0x4],eax
add    eax,0x95
leal   eax, DWORD PTR [edx+0x1f88]
call   0x1f88 <dyld_stub_printf>
mov    DWORD PTR [ebp-0x10],ecx
mov    DWORD PTR [ebp-0x10],ecx
mov    edx,esp
mov    edx,[edx+0x4]
lea    eax,[eax+0x95]
mov    DWORD PTR [edx],eax
mov    reg.c,DWORD PTR [ebp-0x10]  ; printf("%d\n", n);
call   0x1f1b <main+11>
ret
```

```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

struct reg {
    int a,b,c,d;
} reg;

int main(void) {
    int i, n;

    reg.c = a[0];
    reg.c += 1;
    i = reg.c;

    reg.c = i;
    reg.c += 1;
    i = reg.c;

    reg.c = i;
    reg.c += reg.d;

    reg.d = i;
    reg.d += 1;
    i = reg.d;

    reg.d = i;
    reg.d += reg.d;

    reg.c += reg.d;
    n = reg.c;

    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

struct reg {
    int a, b, c, d;
} reg;

int main(void) {
    int i, n;

    reg.c = a[0];
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c = a[reg.c];
    reg.d = i;
    reg.c += reg.d;
    reg.d = i;
    reg.d += 1;
    i = reg.d;
    reg.d = i;
    reg.d = b[reg.d];
    reg.c += reg.d;
    n = reg.c;
    printf("%d\n", n);
}
```
You can take this idea of writing executable pseudo-assembler even further.
include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

struct reg {
    int a,b,c,d;
} reg;

int main(void) {
    int i, n;

    reg.c = a[0];
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c += 1;
    i = reg.c;
    reg.c = i;
    reg.c = a[reg.c];

    reg.d = i;
    reg.c += reg.d;

    reg.d = i;
    reg.d += 1;
    i = reg.d;
    reg.d = i;
    reg.d = b[reg.d];

    reg.c += reg.d;
    n = reg.c;
    printf("%d\n", n);
}
You can take this idea of writing executable pseudo-assembly even further.
You can take this idea of writing executable pseudo-assembler even further.
You can take this idea of writing executable pseudo-assembler even further.
You can take this idea of writing executable pseudo-assembler even further.
You can take this idea of writing executable pseudo-assembler even further.
You can take this idea of writing executable pseudo-assembler even further.
You can take this idea of writing executable pseudo-assembler even further.
You can take this idea of writing executable pseudo-assembler even further.
You can take this idea of writing executable pseudo-assembler even further.
\[ n = i + a[++i] + b[++i]; \]
We have just seen how gcc “interprets” this (meaningless) expression.

\[ n = i + a[++i] + b[+i]; \]
We have just seen how gcc “interprets” this (meaningless) expression.

```c
n = i + a[++i] + b[++i];

tmp.x = a[++i];
tmp.x += i;
tmp.y = b[++i];
n = tmp.x + tmp.y;
```
We have just seen how gcc “interprets” this (meaningless) expression.

\[
n = i + a[++i] + b[+i];
\]

```c
 gcc
tmp.x = a[+i];
tmp.x += i;
tmp.y = b[+i];
n = tmp.x + tmp.y;
```

```c
 gcc
i + a[+i] + b[+i]
i + a[+1] + b[+i]
i + a[2] + b[+i]
i + 4 + b[+i]
2 + 4 + b[+i]
6 + b[+i]
6 + b[+2]
6 + b[3]
6 + 6
12
```
icc
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

The preamble
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

increase the stored value of i to 2. Keep the value 2 in edx.
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

Increase the stored value of \( i \) to \( 3 \). Keep the value \( 3 \) in \( ecx \).
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```


```
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

It looks like icc is first “scanning” through the expression, applying the side effects, and then compute a result.
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

edx is 2, multiply by 4 (sizeof int), use it to index from into array a. Load the value of a[2] into edx. This is just a fancy way of indexing into an array. edx is now 4.
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

```assembly
icc
```
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

- add the stored value of `i` with the evaluated value of `a[++]i`. Notice that `i` is 3 because it has been updated twice already.
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

Add the stored value of `i` with the evaluated value of `a[++i]`. Notice that `i` is 3 because it has been updated twice already.

3 + 4 = 7
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

```asm
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

ecx is 3, multiply by 4 (sizeof int), use it to index from into array b. Load the address of b[3] into ebx.
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

```
add the value of b[3] to edx.
7 + 6 = 13
```
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

Initialize n to 13.
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

Print out and exit.
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

Print out and exit.

```
13
```
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

struct tmp {int x,y,z;} tmp;

int main(void)
{
    int i;
    int n;
    i = a[0] + 1;
    tmp.x = ++i;
    tmp.y = ++i;
    tmp.z = a[tmp.x];
    tmp.z += i;
    tmp.z += b[tmp.y];
    n = tmp.z;
    printf("%d\n", n);
}
```
n = i + a[+i] + b[+i];
n = i + a[++i] + b[++i];

This is how icc interprets this expression
This is how icc interprets this expression

```c
n = i + a[++i] + b[++i];

icc

i = a[0] + 1;
tmp.x = ++i;
tmp.y = ++i;
tmp.z = a[tmp.x];
tmp.z += i;
tmp.z += b[tmp.y];
n = tmp.z;
```
This is how icc interprets this expression

```c
n = i + a[++i] + b[++i];

icc
i = a[0] + 1;
tmp.x = ++i;
tmp.y = ++i;
tmp.z = a[tmp.x];
tmp.z += i;
tmp.z += b[tmp.y];
n = tmp.z;
```

```c
icc
i + a[++i] + b[++i]
i + a[++1] + b[++i]
i + a[2] + b[++i]
3 + 4 + b[3]
7 + b[3]
7 + 6
13
```
clang
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

The preamble
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

```
Evaluate i, save value in register edx
```
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

load value of i into register.
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}

clang

#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}

Load a[2] into register. Note how the old value of i is used to index into array a, it seems like clang indexes from &a[1]. Anyway value of edx was 1. 4 is added. New value is 5.
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

```
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

load i into register, increase and store.
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

edx was 5, now add b[3], and edx is now 11.
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

```
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int a[] = {0, 2, 4, 6, 8};
int b[] = {0, 2, 4, 6, 8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

Clang assembly:
```
0x00001f20 push ebp
0x00001f21 mov ebp,esp
0x00001f23 push edi
0x00001f24 push esi
0x00001f25 sub esp,0x20
0x00001f28 call 0x1f2d <main+13>
0x00001f2d pop eax
0x00001f2e lea ecx,[eax+0x85]
0x00001f34 mov edx,DWORD PTR [eax+0xdf]
0x00001f3a add edx,0x1
0x00001f40 mov DWORD PTR [ebp-0xc],edx
0x00001f43 mov edx,DWORD PTR [ebp-0xc]
0x00001f46 mov esi,DWORD PTR [ebp-0xc]
0x00001f4b mov edi,esi
0x00001f4d add edi,0x1
0x00001f51 mov DWORD PTR [ebp-0xc],edi
0x00001f54 add edx,DWORD PTR [eax+esi*4+0xe3]
0x00001f5b mov esi,DWORD PTR [ebp-0xc]
0x00001f5e mov edi,esi
0x00001f60 add edi,0x1
0x00001f66 mov DWORD PTR [ebp-0xc],edi
0x00001f69 add edx,DWORD PTR [eax+esi*4+0xf7]
0x00001f6f mov DWORD PTR [ebp-0xc],edi
0x00001f70 mov DWORD PTR [ebp-0xc],edx
0x00001f73 mov eax,DWORD PTR [ebp-0x10]
0x00001f76 mov DWORD PTR [esp],ecx
0x00001f79 mov DWORD PTR [esp+0x4],eax
0x00001f7d call 0x1f94 <dyld_stub_printf>
0x00001f82 mov ecx,0x0
0x00001f87 mov DWORD PTR [ebp-0x14],eax
0x00001f8a mov eax,ecx
0x00001f8c add esp,0x20
0x00001f8f pop esi
0x00001f90 pop edi
0x00001f91 pop ebp
0x00001f92 ret
```

Initialize n to 11
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
```c
#include <stdio.h>

int main(void)
{
    int a[] = {0,2,4,6,8};
    int b[] = {0,2,4,6,8};

    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

The C code snippet declares two integer arrays `a` and `b` with the same elements. It then calculates `n = i + a[++i] + b[++i]` where `i` is initialized to 0 and incremented by 1 in each step. Finally, it prints the value of `n`.

The corresponding assembly code is shown on the right side of the image. The highlighted parts in the assembly code correspond to the calculation of `n` and the print statement in the C code.

The assembly code shows how the calculation is performed in hardware, including loading array elements, computing their sum, and printing the result.
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

It seems like clang is doing a typical left-to-right evaluation. Since the C standard does not impose a particular evaluation order, then clang can do whatever it wants, even giving the value that most programmers would expect.
```c
#include <stdio.h>

int a[] = {0,2,4,6,8};
int b[] = {0,2,4,6,8};

struct tmp {int x,y,z;} tmp;

int main(void)
{
    int i;
    int n;

    i = a[0] + 1;
    tmp.x = i;
    tmp.y = i;
    ++i;
    tmp.x += *(a+1 + tmp.y);
    tmp.y = i;
    ++i;
    tmp.x += *(b+1 + tmp.y);

    n = tmp.x;
    printf("%d\n", n);
}
```
\[ n = i + a[+i] + b[+i]; \]
This is how clang interprets this expression

```c
n = i + a[++i] + b[++i];
```
n = i + a[+i] + b[+i];

clang

tmp.x = i;
tmp.y = i;
++i;
tmp.x += *(a+1 + tmp.y);
tmp.y = i;
++i;
tmp.x += *(b+1 + tmp.y);

This is how clang interprets this expression
n = i + a[++i] + b[++i];

clang

tmp.x = i;
tmp.y = i;
++i;
tmp.x += *(a+1 + tmp.y);
tmp.y = i;
++i;
tmp.x += *(b+1 + tmp.y);

clang

i + a[++i] + b[++i]
1 + a[++i] + b[++i]
1 + a[++1] + b[++i]
1 + a[2] + b[++i]
1 + 4 + b[++i]
5 + b[++i]
5 + b[++2]
5 + b[3]
5 + 6
11
How three popular compilers treat an expression with sequence point violation.

\[ n = i + a[++] + b[++] \]

**gcc**

```
tmp.x = a[++i];
tmp.x += i;
tmp.y = b[++];
n = tmp.x + tmp.y;
```

**clang**

```
tmp.x = i;
tmp.y = i;
++i;
tmp.x += *(a + 1 + tmp.y);
tmp.y = i;
++i;
tmp.x += *(b + 1 + tmp.y);
```

**icc**

```
i = a[0] + 1;
tmp.x = ++i;
tmp.y = ++i;
tmp.z = a[tmp.x];
tmp.z += i;
tmp.z += b[tmp.y];
n = tmp.z;
```
This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

```c
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```
Exercise

This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

```c
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

```c
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```
Exercise

This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

```c
#define foo
void foo(void) {
    bool b;
    if (b)   printf("true\n");
    if (!b)  printf("false\n");
}
```

This code is undefined behavior because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

```c
#define bar
void bar(void) {
    char c = 2;
    (void)c;
}
```

```c
#define main
int main(void) {
    bar();
    foo();
}
```
Exercise

This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

```c
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

This is what I get on my computer with no optimization (`-O0 -m32 -mtune=i386`):

```c
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```
Exercise

This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

```c
#include <stdio.h>
#include <stdbool.h>
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

This is what I get on my computer with no optimization (-O0 -m32 -mtune=i386):

```
true
```
Exercise

This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

This is what I get on my computer with no optimization (-O0 -m32 -mtune=i386):

- icc 13.0.1: `true`
- clang 4.1: `false`
Exercise

This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

```c
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

```c
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

This is what I get on my computer with no optimization (`-O0 -m32 -mtune=i386`):

- **icc 13.0.1**
  - `true`
- **clang 4.1**
  - `false`
- **gcc 4.7.2**
  - `true
  - false`
This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

This is what I get on my computer with no optimization (`-00 -m32 -mtune=i386`):

- **icc 13.0.1**
  - `true`
- **clang 4.1**
  - `false`
- **gcc 4.7.2**
  - `true`
  - `false`

(Thanks to Mark Shroyer for blogging about this very interesting gcc behavior [http://markshroyer.com/2012/06/c-both-true-and-false/])
Exercise

This code is **undefined behavior** because b is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

```c
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

This is what I get on my computer with no optimization (−O0 −m32 −mtune=i386):

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>icc 13.0.1</td>
<td>true</td>
</tr>
<tr>
<td>clang 4.1</td>
<td>false</td>
</tr>
<tr>
<td>gcc 4.7.2</td>
<td>true false</td>
</tr>
</tbody>
</table>

with optimization (−O2) I get:

(thanks to Mark Shroyer for blogging about this very interesting gcc behavior [http://markshroyer.com/2012/06/c-both-true-and-false/])
Exercise

This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

```c
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
  bool b;
  if (b)
    printf("true\n");
  if (!b)
    printf("false\n");
}
```

This is what I get on my computer with no optimization (`-O0 -m32 -mtune=i386`):

- icc 13.0.1: `true`
- clang 4.1: `false`
- gcc 4.7.2: `true false`

With optimization (`-O2`) I get:

- `false`

(Thanks to Mark Shroyer for blogging about this very interesting gcc behavior [http://markshroyer.com/2012/06/c-both-true-and-false/](http://markshroyer.com/2012/06/c-both-true-and-false/))
Exercise

This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

This is what I get on my computer with no optimization (`-O0  -m32  -mtune=i386`):

- **icc 13.0.1**
  - True
- **clang 4.1**
  - False
- **gcc 4.7.2**
  - True  false

With optimization (`-O2`) I get:

- **false**
- **false**

(Thanks to Mark Shroyer for blogging about this very interesting gcc behavior [http://markshroyer.com/2012/06/c-both-true-and-false/])
Exercise

This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

```c
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

This is what I get on my computer with no optimization (`-O0 -m32 -mtune=i386`):

- **icc 13.0.1**
  - True
- **clang 4.1**
  - False
- **gcc 4.7.2**
  - True  True

(Thanks to Mark Shroyer for blogging about this very interesting gcc behavior [http://markshroyer.com/2012/06/c-both-true-and-false/](http://markshroyer.com/2012/06/c-both-true-and-false/))
It is looking at assembler time!
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
void foo(void) {
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}

icc 13.0.1 with no optimization (-O0)
void foo(void) {
    char b; // "random" value
    reg.a = b;
    if (reg.a == 0)
        goto label1;
    printf("true\n");
label1:
    reg.a = b;
    if (reg.a != 0)
        goto label2;
    printf("false\n");
label2:
}

icc 13.0.1 with no optimization (-O0)

```
void foo(void) {
    char b; // "random" value
    reg.a = b;
    if (reg.a == 0)
        goto label1;
    printf("true\n");
label1:
    reg.a = b;
    if (reg.a != 0)
        goto label2;
    printf("false\n");
label2:
}
```
void foo(void) {
    bool b;
    if (b) {
        printf("true\n");
    }
    if (!b) {
        printf("false\n");
    }
}

icc 13.0.1 with no optimization (−O0)

void foo(void) {
    char b; // "random" value
    reg.a = b;
    if (reg.a == 0)
        goto label1;
    printf("true\n");
    printf("false\n");
    label1:
        reg.a = b;
    if (reg.a != 0)
        goto label2;
    printf("true\n");
    printf("false\n");
    label2:
    ;
}
void foo(void) {
    bool b;
    if (b)
        printf("true
");
    if (!b)
        printf("false
");
}

icc 13.0.1 with no optimization (-O0)
void foo(void) {
    char b; // "random" value
    reg.a = b;
    if (reg.a == 0)
        goto label1;
    printf("true
");
    label1:
    reg.a = b;
    if (reg.a != 0)
        goto label2;
    printf("false
");
    label2:
}

<table>
<thead>
<tr>
<th>“Random” value</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>anything else</td>
<td>true</td>
</tr>
</tbody>
</table>

icc is doing what most programmers would expect might happen.
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```c
void foo(void) {
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```
void foo(void) {
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}

icc 13.0.1 with optimization (-O2)
void foo(void) {
    reg.a; // "random" value
    if (reg.a != 0)
        goto label1;
    printf("false\n");
    return;
label1:
    printf("true\n");
    return;
}

Notice that icc does not even create space for the variable b. It is just using the random value stored in the eax register.
void foo(void) {
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}

icc 13.0.1 with optimization (-O2)

void foo(void) {
    reg.a; // "random" value
    if (reg.a != 0)
        goto label1;
    printf("false\n");
    return;
label1:
    printf("true\n");
    return;
}

<table>
<thead>
<tr>
<th>“Random” value</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>anything else</td>
<td>true</td>
</tr>
</tbody>
</table>

Notice that icc does not even create space for the variable b. It is just using the random value stored in the eax register.
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
void foo(void) {
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}

void foo(void) {
    char b; // “random” value
    if (((b & 1) != 1)
        goto label1;
    printf("true\n");
    label1:
    if (((b & 1) == 1)
        goto label2;
    printf("false\n");
    label2:
    ;
}
void foo(void) {
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}

clang just tests the last bit of the byte it uses to represent the bool.

clang 4.1 with no optimization (-O0)
void foo(void) {
    char b; // "random" value
    if (((b & 1) != 1)
        goto label1;
    printf("true\n");
label1:  
    if (((b & 1) == 1)
        goto label2;
    printf("false\n");
label2:
    ;
}
void foo(void) {
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}

void foo(void) {
    char b; // "random" value
    if (((b & 1) != 1)
        goto label1;
    printf("true\n");
    label1:
        if (((b & 1) == 1)
            goto label2;
    printf("false\n");
    label2:
}

<table>
<thead>
<tr>
<th>“Random” value</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>even number</td>
<td>false</td>
</tr>
<tr>
<td>odd number</td>
<td>true</td>
</tr>
</tbody>
</table>

clang 4.1 with no optimization (-O0)

clang just tests the last bit of the byte it uses to represent the bool.
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}

clang 4.1 with optimization (-O2)

0x00001f70    push   ebp
0x00001f71    mov    ebp,esp
0x00001f73    sub    esp,0x8
0x00001f76    call   0x1f7b <foo+11>
0x00001f7b    pop    eax
0x00001f7c    lea    eax,[eax+0x37]
0x00001f82    mov    DWORD PTR [esp],eax
0x00001f85    call   0x1f96 <dyld_stub_puts>
0x00001f8a    add    esp,0x8
0x00001f8d    pop    ebp
0x00001f8e    ret
void foo(void) {
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}

void foo(void) {
    puts("false");
}
void foo(void) {
    bool b;
    if (b)
        printf("true\n");
    else
        printf("false\n");
}

clang just prints “false”. Simple and clean!
```c
void foo(void) {
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

```
void foo(void) {
    puts("false");
}
```

```
0x00001f70   push   ebp
0x00001f71   mov    ebp,esp
0x00001f73   sub    esp,0x8
0x00001f76   call   0x1f7b <foo+11>
0x00001f7b   pop    eax
0x00001f7c   lea    eax,[eax+0x37]
0x00001f82   mov    DWORD PTR [esp],eax
0x00001f85   call   0x1f96 <dyld_stub_puts>
0x00001f8a   add    esp,0x8
0x00001f8d   pop    ebp
0x00001f8e   ret
```

```
clang just prints “false”. Simple and clean!
```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
void foo(void) {
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}

gcc 4.7.2 with no optimization (-O0)
void foo(void) {
    char b; // "random" value
    if (b == 0)
        goto label1;
    puts("true");
    label1:
    reg.a = b;
    reg.a ^= 1;
    if (reg.a == 0)
        goto label2;
    puts("false");
    label2:
}

```c
void foo(void) {
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

gcc 4.7.2 with no optimization (-O0)
```c
void foo(void) {
    char b; // "random" value
    if (b == 0)
        goto label1;
    puts("true");

label1:  
    reg.a = b;
    reg.a ^= 1;
    if (reg.a == 0)
        goto label2;
    puts("false");

label2:  
}
```

gcc assumes that the bitpattern in the byte representing a bool is always 0 or 1, never anything else.
void foo(void) {
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}

void foo(void) {
    char b; // "random" value
    if (b == 0)
        goto label1;
    puts("true");
    label1:
    reg.a = b;
    reg.a ^= 1;
    if (reg.a == 0)
        goto label2;
    puts("false");
    label2:
}

gcc 4.7.2 with no optimization (-O0)
gcc assumes that the bitpattern in the byte representing a bool is always 0 or 1, never anything else.

<table>
<thead>
<tr>
<th>“Random” value</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>anything else</td>
<td>true false</td>
</tr>
</tbody>
</table>
void foo(void) {
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}

void foo(void) {
    char b; // "random" value
    if (b == 0)
        goto label1;
    puts("true");
    label1:
        reg.a = b;
        reg.a ^= 1;
    if (reg.a == 0)
        goto label2;
    puts("false");
    label2:
        ;
}

gcc assumes that the bitpattern in the byte representing a bool is always 0 or 1, never anything else.
... and there is nothing wrong with that. We have broken the rules of the language by reading an uninitialized object.
void foo(void) {
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
void foo(void) {
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}

gcc 4.7.2 with optimization (-O2)
void foo(void) {
    puts("false");
}
void foo(void) {
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}

gcc 4.7.2 with optimization (-O2)
void foo(void) {
    puts("false");
}
```c
void foo(void) {
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

### gcc 4.7.2 with optimization (-O2)

```assembly
0x00001edc  push   ebx
0x00001edd  sub    esp,0x18
0x00001ee0  call   0x1ef8 <__x86.get_pc_thunk.bx>
0x00001ee5  lea    eax,[ebx+0x52]
0x00001eeb  mov    DWORD PTR [esp],eax
0x00001eee  call   0x1f14 <dyld_stub_puts>
0x00001ef3  add    esp,0x18
0x00001ef6  pop    ebx
0x00001ef7  ret
```

### gcc 4.7.2 with optimization (-O2)

```c
void foo(void) {
    puts("false");
}
```

**gcc just prints “false”.

<table>
<thead>
<tr>
<th>“Random” value</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>anything else</td>
<td>false</td>
</tr>
</tbody>
</table>
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
Some serious words to wrap it up:

It is common to think that undefined behavior is not such a big deal, and that it is possible to reason about what the compiler might do when encountering code that break the rules. I hope I have illustrated that really strange things can happen, and will happen. It is not possible to generalize about what might happen.

While I don’t show it in this presentation, it is also important to realize that undefined behavior is not only a local problem. The state of the runtime environment will be corrupted, but also the state of the compiler will be corrupted - meaning that UB might result in strange behavior in apparently unrelated parts of the codebase.
But, seriously, who is releasing code with undefined behavior?
But, seriously, who is releasing code with undefined behavior?
But, seriously, who is releasing code with undefined behavior?
But, seriously, who is releasing code with undefined behavior?
But, seriously, who is releasing code with undefined behavior?
But, seriously, who is releasing code with undefined behavior?
But, seriously, who is releasing code with undefined behavior?

snippet from pftn.c in pcc 1.0.0.RELEASE 20110221

... 
/* if both are imag, store value, otherwise store 0.0 */
if (!(li && ri)) {
    tfree(r);
    r = bcon(0);
}
p = buildtree(ASSIGN, l, r);
p->n_type = p->n_type += (FIMAG-FLOAT);
...
But, seriously, who is releasing code with undefined behavior?

Snippet from pftn.c in pcc 1.0.0.RELEASE 20110221

```c
... /* if both are imag, store value, otherwise store 0.0 */
if (!((li && ri)) { 
    tfree(r);
    r = bcon(0);
}
p = buildtree(ASSIGN, l, r);
p->n_type = p->n_type += (FIMAG-FLOAT);
...```

But, seriously, who is releasing code with undefined behavior?

It’s undefined behavior because: “Between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored the you modify and use the value of a variable twice between sequence points.”

snippet from pftn.c in pcc 1.0.0.RELEASE 20110221

```c
/* if both are imag, store value, otherwise store 0.0 */
if (!(li && ri)) {
    tfree(r);
    r = bcon(0);
}
p = buildtree(ASSIGN, l, r);
p->n_type = p->n_type += (FIMAG-FLOAT);
```
In C. Why do you think static variables gets a default value (usually 0), while auto variables does not get a default value?
In C. Why do you think static variables gets a default value (usually 0), while auto variables does not get a default value?

Because C is a braindead programming language?
In C. Why do you think static variables get a default value (usually 0), while auto variables does not get a default value?

Because C is a braindead programming language?

Because C is all about execution speed. Setting static variables to default values is a one time cost, while defaulting auto variables might add a significant runtime cost.
In C. Why is the evaluation order mostly unspecified?
In C. Why is the evaluation order mostly unspecified?
In C. Why is the evaluation order mostly unspecified?

Because C is a braindead programming language?
In C. Why is the evaluation order mostly unspecified?

Because C is a braindead programming language?

Because there is a design goal to allow optimal execution speed on a wide range of architectures. In C the compiler can choose to evaluate expressions in the order that is most optimal for a particular platform. This allows for great optimization opportunities.
Why don’t the C standard require that you always get a warning or error on invalid code?
Why don’t the C standard require that you always get a warning or error on invalid code?

Because C is a braindead programming language?
Why don’t the C standard require that you always get a warning or error on invalid code?

Because C is a braindead programming language?

One of the design goals of C is that it should be relatively easy to write a compiler. Adding a requirement that the compilers should refuse or warn about invalid code would add a huge burden on the compiler writers.
The spirit of C

**trust the programmer**
- let them do what needs to be done
- the programmer is in charge not the compiler

**keep the language small and simple**
- small amount of code $\rightarrow$ small amount of assembler
- provide only one way to do an operation
- new inventions are not entertained

**make it fast, even if its not portable**
- target efficient code generation
- int preference, int promotion rules
- sequence points, maximum leeway to compiler

**rich expression support**
- lots of operators
- expressions combine into larger expressions