Consequences of bugs:

- compiler gives syntax/semantic error - *if you're very lucky*
- program halts with run-time error - *if you're lucky*
- program never halts - *if you're lucky-ish*
- program halts, but with incorrect results - *if you're unlucky*
- program appears correct, but has security holes - *if you're unlucky*

Invalid C Program - changed variable

```c
int a[10];
int b[10];
printf("a[0] is at address %p\n",&a[0]);
printf("a[9] is at address %p\n", &a[9]);
printf("b[0] is at address %p\n",&b[0]);
printf("b[9] is at address %p\n", &b[9]);
for (int i = 0; i < 10; i++) {
    a[i] = 77;
}
for (int i = 0; i <= 12; i++) {
    b[i] = 42;
}
for (int i = 0; i < 10; i++) {
    printf("%d ", a[i]);
}
printf("\n");
```

Invalid C Program - changed variable

The C program assigns to b[10] .. b[12] which don't exist

The consequence could be anything - a C implementation is permitted to behave in any manner given an invalid program.

On gcc 6.3 on Linux/x86_64 it happens to change b[0] to 42:

```
$ gcc invalid_array_index0.c
$ a.out
a[0] is at address 0x7fffc9cbcbf0
a[9] is at address 0x7fffc9cbcc14
b[0] is at address 0x7fffc9cbcbc0
b[9] is at address 0x7fffc9cbcbe4
42 77 77 77 77 77 77 77 77 77
```

Invalid C Programs - changed termination

```c
int i;
int a[10];
printf("i is at address %p\n", &i);
printf("a[0] is at address %p\n", &a[0]);
printf("a[9] is at address %p\n", &a[9]);
printf("a[11] would be stored at address %p\n", &a[10]);

for (i = 0; i <= 11; i++) {
    a[i] = 0;
}
```

Invalid C Programs - changed termination

Another invalid C program assigning to a non-existent array element.
On gcc 6.3 on Linux/x86_64 it happens to assigns to i and the loop doesn't terminate.
So a one character error makes the program invalid, and seemingly certain termination does not occur.

```
$ gcc invalid1.c
$ a.out
i is at address 0x7fffbb72bfdc
a[0] is at address 0x7fffbb72bfb0
a[9] is at address 0x7fffbb72bfd4
a[10] is equivalent to address 0x7fffbb72bfd8
....
```

Invalid C Program - changed variable in another function

```
int main(void) {
    int answer = 36;
    f(5);
    printf("answer=%d\n", answer); // prints 42
    return 0;
}
void f(int x) {
    int a[10];

    // a[19] doesn't exist
    // with gcc 6.3 on Linux/x86_64 variable answer
    // in main happens to be where a[19] would be
    a[19] = 42;
}
```

Invalid C Program - changed variable in another function

Yet another invalid C program assigning to a non-existent array element.
On gcc 6.3 on Linux/x86_64 it changes the variable answer in the calling function main.

```
$ gcc invalid2.c
$ a.out
answer=42
```

Invalid C Program - changed function return location

```c
void f() {
    int a[10];
    // on gcc-6.3/Linux x86
    // change function's return address on stack
    // causing function to return after the line
    // where answer is assigned 24
    a[14] += 7;
}
int main(void) {
    int answer = 42;
    f();
    answer = 24;
    printf("answer=%d\n", answer);
    return 0;
}
```

Invalid C Program - changed function return location

Yet another invalid C program assigning to a non-existent array element.
With gcc 6.3 on Linux/x86_64 it changes where the function returns in main.

```
$ gcc invalid3.c
$ a.out
answer=42
```

Invalid C Program - bypassing authentication

```c
int authenticated = 0;
char password[8];
printf("Enter your password: ");
gets(password);
if (strcmp(password, "secret") == 0) {
    authenticated = 1;
}
// a password longer than 8 characters will overflow
// array password  on gcc 6.3 on Linux/x86_64 this can
// overwrite the variable authenticated and allow access
if (authenticated) {
    printf("Welcome. You are authorized.\n");
} else {
    printf("Welcome. You are unauthorized. ");
    printf("Your death will now be implemented.\n");
    printf("Welcome. You will experience ");
    printf("a tingling sensation and then death. \n");
    printf("Remain calm while your life is extracted.\n");
}
```

Invalid C Program - bypassing authentication

Yet another invalid C program assigning to a non-existent array element.
A password longer than 8 characters will overflow the array password This is often turmed **buffer-overflow**.

```
$ gcc invalid4,c
$ a.out
Enter your password: secret
Welcome. You are authorized.
$ a.out
Enter your password: wrong
Welcome. You are unauthorized.
Your death will now be implemented.
Welcome. You will experience a
tingling sensation and then death.
Remain calm while your life is extracted.
$ a.out
Enter your password: longcorrectpassword
Welcome. You are authorized.
```

Implementation versus Language

C was designed for much smaller slower computers - 28K of RAM , 1mhz clock.
Program speed/size much more important for programs then dominated language choice.
Most C implementations still focus on maximizing performance of valid programs.
Most C implementations do not check array bounds or for arithmetic overflow because this has performance costs.
The C definition does not entail this.
A C implementation an check array bounds and halt if an invalid indexes is used.
A C implementation could check & halt if an unininitialized value is used - but difficult/expensive to track for arrays.

Address Sanitizer extension to gcc/clang

gcc -fsanitize=address gives a very different Cimplementation.
Invalid array indices, pointer dereferences and some other invalid use of the string library function are detected.
Performance cost - execution from 1.2-10+x slower.
Information cryptic but note source code line indicated, e.g.:

```
$ cd /home/cs1511/public_html/lec/illegal_C/code/
$ gcc -g -fsanitize=address debug_examples.c
$ ./a.out 3
ASAN:DEADLYSIGNAL
============================
==16917==ERROR: AddressSanitizer: SEGV on unknown address
0x000000000014 (pc 0x55819087cd2c bp 0x7ffd02a40bb0 ....
    #0 0x55819087cd2b in test3 debug_examples.c:33
    #1 0x55819087d19c in main debug_examples.c:96
    #2 0x7fccf078d2b0 in __libc_start_main (/lib/...
    #3 0x55819087caf9 in _start ...
....
```

Address Sanitizer extension to gcc/clang

dcc uses -fsanitize=address (with clang) but makes message more comprehensible for beginner programmers:

```
$ cd /home/cs1511/public_html/lec/illegal_C/code/
$ dcc debug_examples.c
$ ./a.out 3
ASAN:DEADLYSIGNAL

debug_examples.c:33 runtime error - illegal array, pointer
or other operation
Execution stopped in test3() in debug_examples.c line 33:

    int *a = NULL;
    // dereferencing NULL pointer
--> a[5] = 42;
}


Values when execution stopped:
```

Address Sanitizer extension to gcc/clang

Address Sanitizer does not detect use of uninitialized values, e.g.:

```
% ./debug_examples 4
0
1
2
3
-2115323248
5
6
7
8
9
```

valgrind - another debugging/testing tool

Valgrind works on x86 machine code - not C specific.
Valgrind runs the code on a virtual machine and detects use of uninitialized memory.
Also picks up many invalid array indexes and pointer dereferences:
Large performance penalty - and slow start time.

valgrind - another debugging/testing tool

```
% valgrind ./debug_examples 4
==1932== Memcheck, a memory error detector
==1932== Copyright (C) 2002-2010, and GNU GPL'd, ...
==1932== Using Valgrind-3.6.1 and LibVEX; rerun ...
==1932== Command: ./debug_examples 4
==1932==
0
1
2
3
==1932== Use of uninitialised value of size 8
==1932==    at 0x521AF0B: _itoa_word (_itoa.c:195)
==1932==    by 0x521D3B6: vfprintf (vfprintf.c:1619)
==1932==    by 0x400FBF: test4 (debug_examples.c:45)
==1932==    by 0x401317: main (debug_examples.c:92)
==1932==
...
```

dcc –valgrind

dcc –valgrind causes valgrind to used to run yuor program,makes messages more comprehensible for beginner programmers: transbe run
For example:

```
$ dcc --valgrind debug_examples.c
% ./a.out 4
Runtime error: uninitialized variable accessed.
Execution stopped in test4() debug_examples.c line 45:

    // accessing uninitialized array element (a[4])
    for (i = 0; i < 10; i++)
-->     printf("%d\n", a[i]);
}

Values when execution stopped:
a = {0, 1, 2, 3, -16776544, 5, 6, 7, 8, 9}
i = 4
a[i] = -16776544
```