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*

Perl and Language Safety

We've seen Perl's design introduces quite a few possibilities for bugs and security holes.
For contrast let examine the design of C particularly wrt invalid programs.

Invalid C Program - changed variable

/home/cs2041/public_html/code/safety/invalid0.c

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

Invalid C Program - changed variable

The C program assigns to a[10] which does not exist.
The consequence could be anything - a C implementation is permitted to behave in any manner given an invalid program.
On gcc-5.3/x86 it happens to change b[0] to 42:

```
$ gcc invalid_array_index0.c
$ a.out
i is at address 0xbffff65c
a[0] is at address 0xbffff634
a[9] is at address 0xbffff658
b[0] is at address 0xbffff60c
b[9] is at address 0xbffff630
42 77 77 77 77 77 77 77 77 77
```

Invalid C Programs - changed termination

/home/cs2041/public_html/code/safety/invalid1.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[10] is equivalent to address %p\n", &a[10]);
for (i = 0; i <= 10; i++)
    a[i] = 0;
```

Invalid C Programs - changed termination

Another invalid C program assigning to a non-existent array element.
With gcc-5.3/x86 it 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 0xbffff65c
a[0] is at address 0xbffff634
a[9] is at address 0xbffff658
a[10] is equivalent to address 0xbffff65c
```

Invalid C Program - changed variable in another function

/home/cs2041/public_html/code/safety/invalid2.c

```
void f(int x) {
    int a[10];
    a[19] = 42; /* change variable answer in main (gcc 4.9/
}
int main(void) {
    int answer = 36;
    f(5);
    printf("answer=%d\n", answer);
    return 0;
}
```

Invalid C Program - changed variable in another function

Yet another invalid C program assigning to a non-existent array element.
With gcc-5.3/x86 it changes the variable answer in the calling function main.

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

Invalid C Program - changed function return location

/home/cs2041/public_html/code/safety/invalid3.c

```
void f() {
    int a[10];
    // change function's return address on stack (gcc-5.3/I
    // causing function to return after the line:  answer =
    a[11] += 10;
}
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-5.3/x86 it changes the variable answer in the calling function main.

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

Invalid C Program - bypassing authentication

/home/cs2041/public_html/code/safety/invalid4.c

```
int authenticated = 0;
char password[8];
printf("Enter your password: ");
gets(password);
if (strcmp(password, "secret") == 0)
    authenticated = 1;
if (authenticated) {
    printf("Welcome. You are authorized.\n");
} else {
    printf("Welcome. You are unauthorized. Your death w
    printf("Welcome. You will experience a tingling sen
    printf("Remain calm while your life is extracted.\r
}
```

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 on gcc-5.1 x86/Linux this can change the variable authenticated and allow access without knowing the correct password.
This is often turmed **buffer-overflow**.

```
$ a.out
Enter your password: secret
Welcome. You are authorized.
$ a.out
Enter your password: hello
Welcome. You are unauthorized. Your death will now be imple
Welcome. You will experience a tingling sensation and then
Remain calm while your life is extracted.
```

Invalid C Program - unexpected code excution

/home/cs2041/public_html/code/safety/invalid5.c

```
char *machine_instructions = "\x31\xc0\x50\x68\x2f\x2f\x73\

void f() {
    int a[10];
    printf("Running a shell\n");
    a[13] = (int)machine_instructions;
}

int main(void) {
    f();
    return 0;
}
```

Invalid C Program - unexpected code execution

On gcc-5.1 x86/Linux the illegal access will cause execution the contents of the array **machine_instructions**
The machine code in this array run a shell.

```
$ gcc invalid5.c
$ a.out
Running a shell
sh-4.3$
```

This is often a key part of security exploits.

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 (like Java) can check array bounds and halt if an invalid indexes is used.

A C implementation could check & halt if an unititialized value is used - but difficult/expensive to track for arrays.

An Invalid C Programs

This C program executed various pieces of invalid code depending on its argument:

/home/cs2041/public_html/code/safety/debug_examples.c

```
#include <stdio.h>
#include <stdlib.h>

// detected by gcc -O -Wall
// detected by valgrind
void test0(void) {
    int i;
    // accessing uninitialized local variable
    printf("%d\n", i);
}

// detected by gcc -fsanitize=address
// detected by valgrind
void test1(void) {
    int *a = malloc(10*sizeof (int));
    // accessing variable outside malloc'ed region
    a[10] = 42;
}

// detected by gcc -fsanitize=address
// detected by valgrind
void test2(void) {
    int *a = malloc(10);
    // accessing variable outside malloc'ed region
    a[5] = 42;
}

// detected by gcc -fsanitize=address
// detected by valgrind
void test3(void) {
    int *a = NULL;
    // dereferencing NULL pointer
    a[5] = 42;
}

// detected by valgrind
void test4(void) {
    int i;
    int a[10];
    for (i = 0; i < 10; i++)
        if (i != 4)
            a[i] = i;
    // accessing uninitialized array element (a[4])
    for (i = 0; i < 10; i++)
        printf("%d\n", a[i]);
}

// detected by valgrind
void test5(void) {
    int i;
    int *a = malloc(10*sizeof (int));
    for (i = 0; i < 10; i++)
        if (i != 4)
            a[i] = i;
    // accessing uninitialized part of malloc'ed region
    for (i = 0; i < 10; i++)
        printf("%d\n", a[i]);
}

// detected by gcc -fsanitize=address
// detected by valgrind
void test6(void) {
    int *p = malloc(10*sizeof *p);
    int *q = p;
    free(p);
    // accessing free'd area
    q[4] = 42;
}

// detected by valgrind '--leak-check=yes'
void test7(void) {
    int *p = malloc(10*sizeof *p);
    // malloc'ed space isn't freed
}

// detected by gcc  -fsanitize=address
// detected by valgrind
void test8(void) {
    int *p = malloc(10*sizeof *p);
    int *q = p;
    free(p);
    // multiple free
    free(q);
}
int g[10];

void test9(void) {
    g[10] = 0;
}

int main(int argc, char*argv[]) {
    switch (atoi(argv[1])) {
    case 0: test0(); break;
    case 1: test1(); break;
    case 2: test2(); break;
    case 3: test3(); break;
    case 4: test4(); break;
    case 5: test5(); break;
    case 6: test6(); break;
    case 7: test7(); break;
    case 8: test8(); break;
    case 9: test9(); break;
    }
    return 0;
}
```

address sanitizer extension to gcc/clang

gcc -fsanitize=address is a very different C implementation.
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/cs2041/public_html/lec/safety/examples/
% gcc -g -fsanitize=address  debug_examples.c -o debug_examples
% ./debug_examples 3
==5780==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xb5900804 at ...
    #0 0x804.94b in test2 /home/cs2041/public_html/15s2/code/safety/debug_examples.c:25
    #1 0x8048cfc in main /home/cs2041/public_html/15s2/code/safety/debug_examples.c:95
.....
```

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 poointer dereferences:
For example:

```
% valgrind ./debug_examples 4
==1932== Memcheck, a memory error detector
==1932== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==1932== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==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 0x4E3DC8D: __mfwrap_printf (in /usr/lib/x86_64-linux-gnu/libmudflap.s
==1932==    by 0x400FBF: test4 (debug_examples.c:45)
==1932==    by 0x401317: main (debug_examples.c:92)
==1932==
...
```

Other tools

Stack protection (canaries) e.g. gcc -fstack-protector-all.
Debugging malloc libraries - dmalloc and efence.
Commercial : coverity and purify - different (complementary) approach to valgrind.