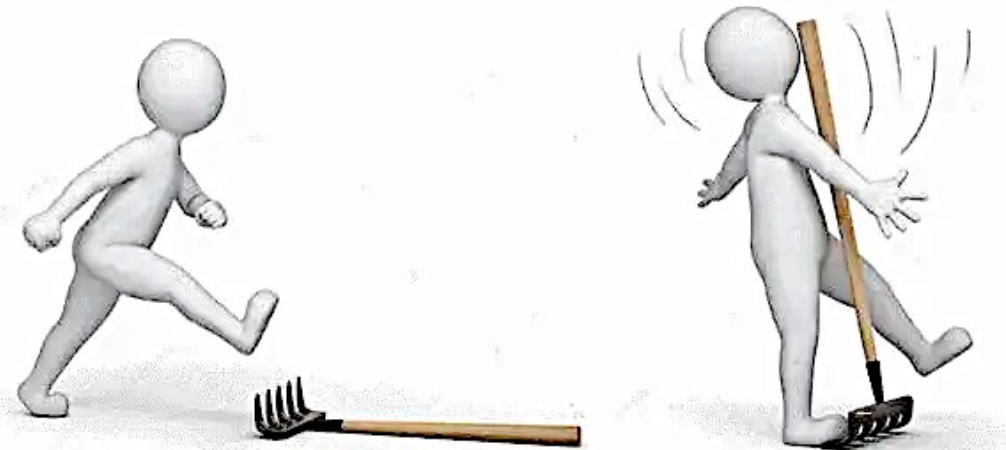
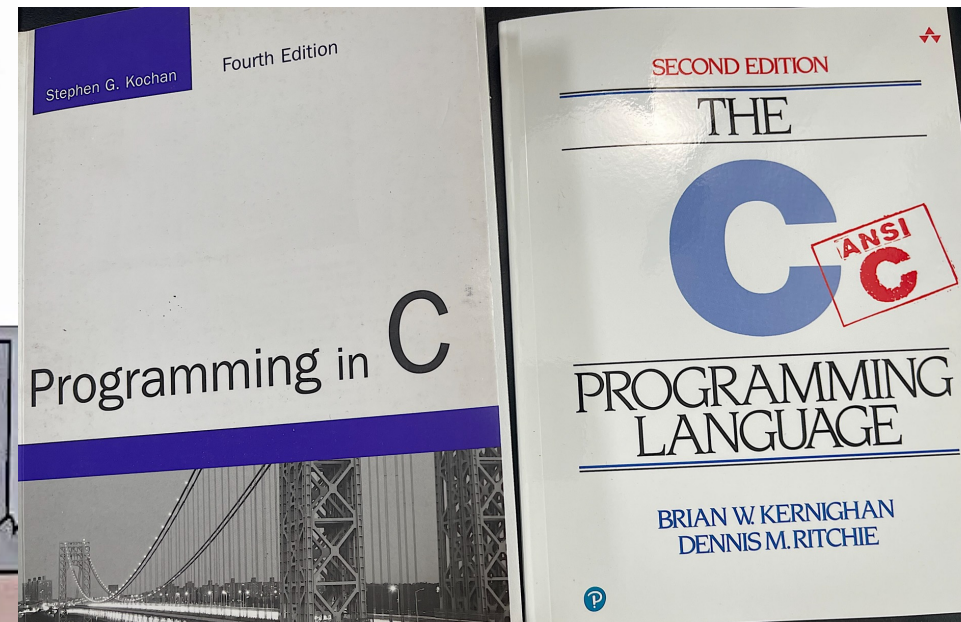


Lecture 9: C part II



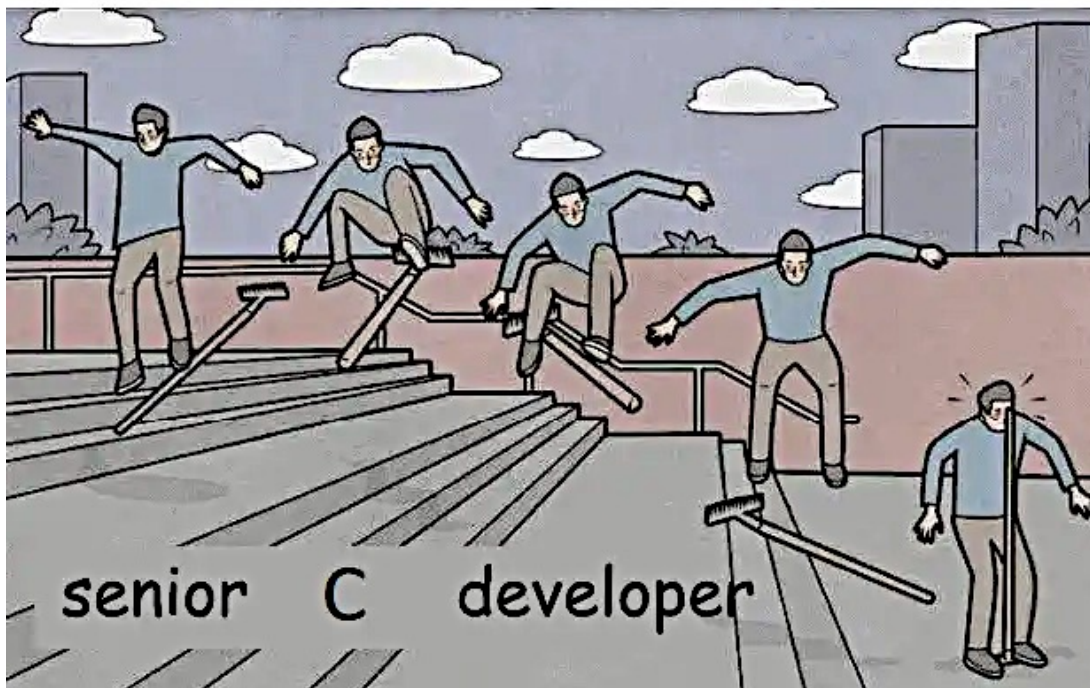
junior C developer

Suggested Literature



Suggested Literature

Both books are available at
<https://learning.oreilly.com>



senior C developer

RECAP: Hello World in C

Hello World has many different features compared to **bash**

```
GNU nano 4.8 helloworld.c
#include <stdio.h>

int main (void)
{
    printf ("Physics 504 is fun.\n");
    return 0;
}
```

Arguments of main allow command line arguments to be passed to C

printf here is similar in many ways to Bash

RECAP: Comments and variables in C

Two commands for comments in C : `///` and `/* ... */`

```
GNU nano 4.8                                sumint.c
#include <stdio.h>

/*
Multi line observations
Line 2!!
*/

int main (void)
{
    int value1, value2;

    value1 = 50;
    value2 = 25;
    int sum = value1 + value2; // you can declare inline

    printf ("The sum of %i and %i is %i\n", value1, value2, sum);

    return 0;
}
```

whovian@phy504:~/host/C\$ gcc sumint.c -o sum

whovian@phy504:~/host/C\$./sum

The sum of 50 and 25 is 75

**Comments
(not compiled)**



RECAP: float/double variables in C

```
#include <stdio.h>

int main (void)
{
    // float has 10^-7 prec. max
    float a = 1.0012033;
    float b = 1.0012034;
    float c = b - a;

    printf ("c = %g\n", c);

    // float has 10^-14 prec. max
    double f = 1.0012033;
    double g = 1.0012034;
    double h = g - f;

    printf ("h = %g\n", h);

    return 0;
}
```

“Problem” in EVERY language

[Good discussion on StackOverflow on float vs double vs long double](#)

[What Every Computer Scientist Should Know About Floating-Point Arithmetic](#)

```
whovian@phy504:~/host/C$ nano floatprec.c
whovian@phy504:~/host/C$ gcc floatprec.c -o float
whovian@phy504:~/host/C$ ./float
c = 1.19209e-07
h = 1e-07
whovian@phy504:~/host/C$
```

wrong result

RECAP: Assignment Operators

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

```
int main (void)  
{
```

```
    int a = 10;
```

```
    printf ("a = %i\n", a);
```

```
    a += 10;
```

```
    printf ("a = %i\n", a);
```

```
    a *= 8;
```

```
    printf ("a = %i\n", a);
```

```
    a /= 4;
```

```
    printf ("a = %i\n", a);
```

```
    return 0;
```

```
}
```

C allows joint arithmetic operators with assignment

$a = a + 10$

$a = a * 8$

$a = a / 4$

```
whovian@phy504:~/host/C$ nano aoper.c
```

```
whovian@phy504:~/host/C$ gcc aoper.c -o aoper
```

```
whovian@phy504:~/host/C$ ./aoper
```

```
a = 10
```

```
a = 20
```

```
a = 160
```

```
a = 40
```

RECAP: Be careful with **unsigned int**



Programmer Humor
@PROGRAMMERHUMOR

This for loop wreaked havoc in our code for some time [reddit.com/r/programmerhu...](https://www.reddit.com/r/programmerhumor/)

```
int main()
{
    unsigned int i = 0;

    for(i = 10; i >= 0; i--) {
        printf("%d\n", i);
    }

    return 0;
}
```

6:00 PM · 1/26/23 · 124K Views

This loop runs forever
(try it yourself)!

There is no negative
sign w/ **unsigned int**

When doing **i--** on **i=0**
iteration, the variable
underflows (assigned
garbage, but positive
garbage!)

RECAP: Loops (**for**) w/ **scanf**

scan for input

```
#include <stdio.h>

int main (void)
{
    int m;
    printf ("What triangular number do you want? ");
    scanf ("%i", &m);

    int tn = 0;
    for (int n = 1; n < m+1; ++n)
    {
        tn += n;
    }

    printf ("Triangular number %i is %i\n", m, tn);
    return 0;
}
```

whovian@phy504:~/host\$ gcc printtrinumloop2.c -o printtrinumloop2
whovian@phy504:~/host\$./printtrinumloop2
What triangular number do you want? 10
Triangular number 10 is 55

Reminder: don't use short cryptic names
(I am using it here to fit code in a slide)

&x is the **reference** to variable x
(basically its memory position)
so **scanf** knows where to write it

Reminder: many books don't do that, but
you can (and should) define variables as
local as possible (here inside the loop)

Loops (**while**)

```
#include <stdio.h>
int main (void)
{
    int ct = 1;

    while (ct < 6)
    {
        printf ("First Run: %i\n", ct);
        ++ct;
    }
    // while may never run!
    while (ct < 6)
    {
        printf ("Second run: %i\n", ct);
        ++ct;
    }
    return 0;
}
```

whovian@phy504:~/host\$ gcc whileloops.c -o whileloops
whovian@phy504:~/host\$./whileloops
First Run: 1
First Run: 2
First Run: 3
First Run: 4
First Run: 5

If the condition fails right away, the program skips over the code inside the brackets {}

Loops (**while**) – interesting case

```
#include <stdio.h>
int main()
{
    printf( "\t--- Calculate Averages ---\n" );
    printf( "Enter some small numbers (type a letter to end input) \n" );

    double x;
    double sum = 0;
    int count = 0;

    while (scanf("%lf", &x) == 1)
    {
        printf("x = %lf\n", x);
        sum += x;
        ++count;
    }
    if (count == 0)
    {
        printf("No input data!\n" );
    }
    else
    {
        printf("Average = %.2f\n", sum/count );
    }
    return 0;
}
```

scanf does return an integer!
scanf returns the number of
fields that were successfully
converted and assigned



Loops (**while**) – interesting case

```
whovian@phy504:~/host$ ./whileloops2
    --- Calculate Averages ---
Enter some small numbers (type a letter to end input)
1 2 3 4
x = 1.000000
x = 2.000000
x = 3.000000
x = 4.000000
5678
x = 5678.000000
2 4 5
x = 2.000000
x = 4.000000
x = 5.000000
X
Average = 712.38
```

The while loop produces an interesting behavior. As long as you write numbers, the program keeps waiting for more input

Loops (**do-while**)

```
#include <stdio.h>

int main (void)
{
    int ct = 1;

    do
    {
        printf ("First Run: %i\n", ct);
        ++ct;
    } while (ct < 6);

    do
    {
        printf ("Second run: %i\n", ct);
        ++ct;
    } while (ct < 6);

    return 0;
}
```

whovian@phy504:~/host\$ gcc dowhileloops.c -o dowhileloops
whovian@phy504:~/host\$./dowhileloops
First Run: 1
First Run: 2
First Run: 3
First Run: 4
First Run: 5
Second run: 6

← we need a semicolon here

← **do{...}while** loops always run at least once!

if / else statements

```
#include <stdio.h>

int main (void)
{
    int num;
    printf ("Type in your number: ");
    scanf ("%i", &num);

    if (num < 0)
    {
        num = -num;
    }

    printf ("The absolute value is %i\n", num);
    return 0;
}
```

You don't need to have the
else{...} block if not
needed

if / else statements

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

```
int main ()
```

```
{
```

```
    int num;
```

```
    printf ("Enter your number to be tested: ");
```

```
    scanf ("%i", &num);
```

```
    int rem = num % 2;
```

```
    if (rem == 0)
```

```
    {
```

```
        printf ("The number is even.\n");
```

```
    }
```

```
    else
```

```
    {
```

```
        printf ("The number is odd.\n");
```

```
    }
```

```
    return 0;
```

```
}
```

```
whovian@phy504:~/host$ ./ifelse2
Enter your number to be tested: 4
The number is even.
whovian@phy504:~/host$ ./ifelse2
Enter your number to be tested: 5
The number is odd.
```

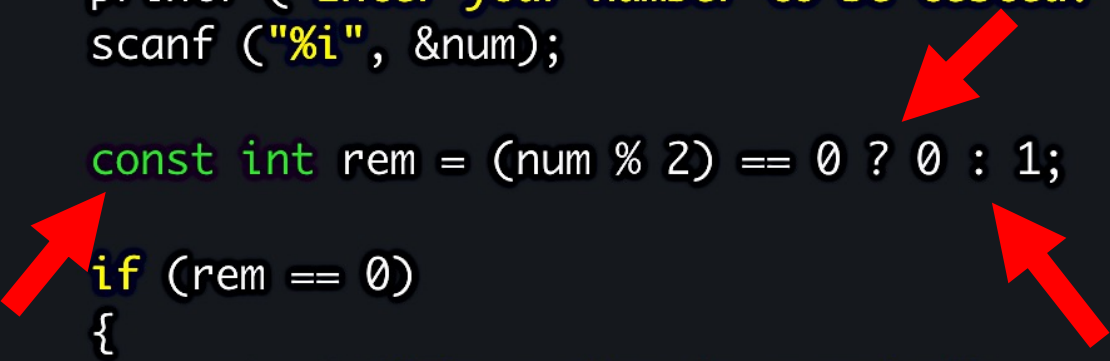
if / else statement in one line (quite useful)

```
#include <stdio.h>

int main ()
{
    int num;
    printf ("Enter your number to be tested: ");
    scanf ("%i", &num);

    const int rem = (num % 2) == 0 ? 0 : 1;

    if (rem == 0)
    {
        printf ("The number is even.\n");
    }
    else if (rem == 1)
    {
        printf ("The number is odd.\n");
    }
    else
    {
        printf("This should be impossible\n");
    }
    return 0;
}
```



This is the only way to use **if/else** to define constant variables

This code also illustrates how to write code with **if / else if / else**

Here is the first time I am going to talk about “**const correctness**”

if / else statement in one line (useful!)

```
GNU nano 4.8 ifelse5.c
#include <stdio.h>

int main ()
{
    int num;
    printf ("Enter the year to be tested: ");
    scanf ("%i", &num);

    const int res = (num < 0) ? -1 : (num == 0) ? 0 : 1;

    printf ("The sign of the number is %i \n", res);

    return 0;
}
```

You can have nested on-line conditional statements (? :)

```
whovian@phy504:~/host$ ./ifelse5
Enter the year to be tested: 200
The sign of the number is 1
whovian@phy504:~/host$ ./ifelse5
Enter the year to be tested: 0
The sign of the number is 0
whovian@phy504:~/host$ ./ifelse5
Enter the year to be tested: -100
The sign of the number is -1
```

if / else statement with logical operators

```
#include <stdio.h>

int main ()
{
    int num;
    printf ("Enter the year to be tested: ");
    scanf ("%i", &num);

    const int rem4 = num % 4;
    const int rem100 = num % 100;
    const int rem400 = num % 400;

    if ((rem4 == 0 && rem100 != 0) || rem400 == 0)
    {
        printf ("It is a leap year.\n");
    }
    else
    {
        printf ("It is not a leap year.\n");
    }
    return 0;
}
```

General advice: if
a variable is not
going to be
modified, make it
const!

That helps with
debugging and
optimization.

```
whovian@phy504:~/host$ gcc ifelse4.c -o ifelse4
whovian@phy504:~/host$ ./ifelse4
Enter the year to be tested: 1986
It is not a leap year.
```

const correctness in a code

If a variable is not going to be modified, make it **const**!

16 Answers

Sorted by: Highest score (default)



This is the definitive article on "const correctness": <https://isocpp.org/wiki/faq/const-correctness>.

191

In a nutshell, using const is good practice because...



0. It protects you from accidentally changing variables that aren't intended be changed,
1. It protects you from making accidental variable assignments. For instance, you are protected from

```
if( x = y ) // whoops, meant if( x == y ).
```

2. The compiler can optimize it.

At the same time, the compiler can generate more efficient code because it knows exactly what the state of the variable/function will be at all times. If you are writing tight C++ code, this is good.

You are correct in that it can be difficult to use const-correctness consistently, but the end code is more concise and safer to program with. When you do a lot of C++ development, the benefits of this quickly manifest.

Share Edit Follow Flag

edited Jan 12, 2023 at 0:28



SKPS

5,835 ● 5 ● 36 ● 72

answered Sep 25, 2008 at 23:40



Jordan Parmer

37.2k ● 30 ● 99 ● 120

<https://stackoverflow.com/a/136917/2472169>

const correctness in a code

If a variable is not going to be modified, make it **const**!



144



Here's a piece of code with a common error that const correctness can protect you against:

```
void foo(const int DEFCON)
{
    if (DEFCON = 1)    //< FLAGGED AS COMPILER ERROR! WORLD SAVED!
    {
        fire_missiles();
    }
}
```

Share Edit Follow Flag

edited Aug 29, 2010 at 16:18

answered Sep 25, 2008 at 23:37



Doug T.

65.6k ● 28 ● 141 ● 205

<https://stackoverflow.com/a/136895/2472169>

- 40 ▲ Of course, most modern compilers warn about assignments in conditionals and we all turn on the treat warnings as errors flag, right :-> – Jack Bolding Sep 26, 2008 at 1:06
- 10 ▲ Don't be too hard on that example: This is the same example evangelized by some developers to convince us to use `if(0 == i)` instead of `if(i == 0)`. At last, Doug's pattern of code can be used outside the "if" statement. What's important is it shows one of the benefits of const in a humorous way. + 1. – paercebal Oct 11, 2008 at 20:45
- 2 ▲ Some compilers (and lints) have been warning about this for a long time, and that is much more useful to catch this typo than const: codepad.org/Nnf5JUXV. – Roger Pate Mar 16, 2010 at 5:14
- 9 ▲ @Roger you assume that we live in a world where builds are clean and warning free. My experience is that in a lot of code warnings get lost in the noise. Also, there is a lot of code that does assignments in the if expression and many will argue that it is a valid, "good" style. – Doug T. Mar 16, 2010 at 15:51
- 2 ▲ @Doug: This is not about style, as I use this pattern myself: you can make it obvious with `if ((a = b))`, which also makes it more readable. I won't open the debate about compiler warnings here, but I will point out that various lint tools are more configurable (so no noise), don't require you to change your compiler, and a complete solution to the problem of `==/=` typos, while using const only catches some of them. – Roger Pate Mar 17, 2010 at 1:37

Arrays

There are two ways to define arrays in C. Today we will learn one of them

GNU nano 4.8

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

```
int main (void)
```

```
{
```

```
int x[5];
```

size of the array

counting start from zero

```
x[0] = 197;
```

```
x[1] = 200;
```

```
x[2] = -100;
```

```
x[3] = x[0] + x[1];
```

```
x[4] = 350;
```

```
whovian@phy504:~/host$ ./arrays
```

```
x[0] = 197
```

```
x[1] = 200
```

```
x[2] = -100
```

```
x[3] = 397
```

```
x[4] = 350
```

```
for (int i=0; i<5; ++i)
```

```
{
```

```
    printf ("x[%i] = %i\n", i, x[i]);
```

```
}
```

```
return 0;
```

```
}
```

Arrays

Array **slicing** goes from **zero** to **size - 1**

```
#include <stdio.h>

int main (void)
{
    const int size = 5;
    int x[size];

    x[0] = 197;
    x[1] = 200;
    x[2] = -100;
    x[3] = x[0] + x[1];
    x[4] = 350;

    for (int i=0; i<size; ++i)
    { // in C - arrays goes from zero to size - 1
        printf ("x[%i] = %i\n", i, x[i]);
    }
    return 0;
}
```

< size (this means last element is size-1)



Arrays

The size of the array can be chosen at runtime as well!

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

```
int main (void)  
{
```

```
    int size;
```

```
    scanf ("%i", &size);
```

```
    int x[size];
```

```
    for (int i=0; i<size; ++i)
```

```
    { // in C - arrays goes from zero to size - 1
```

```
        x[i] = 2* i;
```

```
        printf ("x[%i] = %i\n", i, x[i]);
```

```
    }
```

```
    return 0;
```

```
}
```

compiler does not know the
size of the array!

This is called

Variable Length

Arrays (caution: optional
feature in C11)

```
whovian@phy  
whovian@phy  
10  
x[0] = 0  
x[1] = 2  
x[2] = 4  
x[3] = 6  
x[4] = 8  
x[5] = 10  
x[6] = 12  
x[7] = 14  
x[8] = 16  
x[9] = 18
```

[Be careful with large arrays](#) (link from StackOverflow about StackOverflow!)

Arrays

Alternative way to initialize arrays

```
#include <stdio.h>

int main (void)
{
    int x[10] = {0, 1, 4, 9, 16};

    for (int i=5; i<10; ++i)
    {
        x[i] = i * i;
    }

    for (int i=0; i<10; i++)
    {
        printf ("x[%i] = %i\n", i, x[i]);
    }

    return 0;
}
```



```
whovian@ph
whovian@ph
x[0] = 0
x[1] = 1
x[2] = 4
x[3] = 9
x[4] = 16
x[5] = 25
x[6] = 36
x[7] = 49
x[8] = 64
x[9] = 81
```

Multi-dimensional Arrays

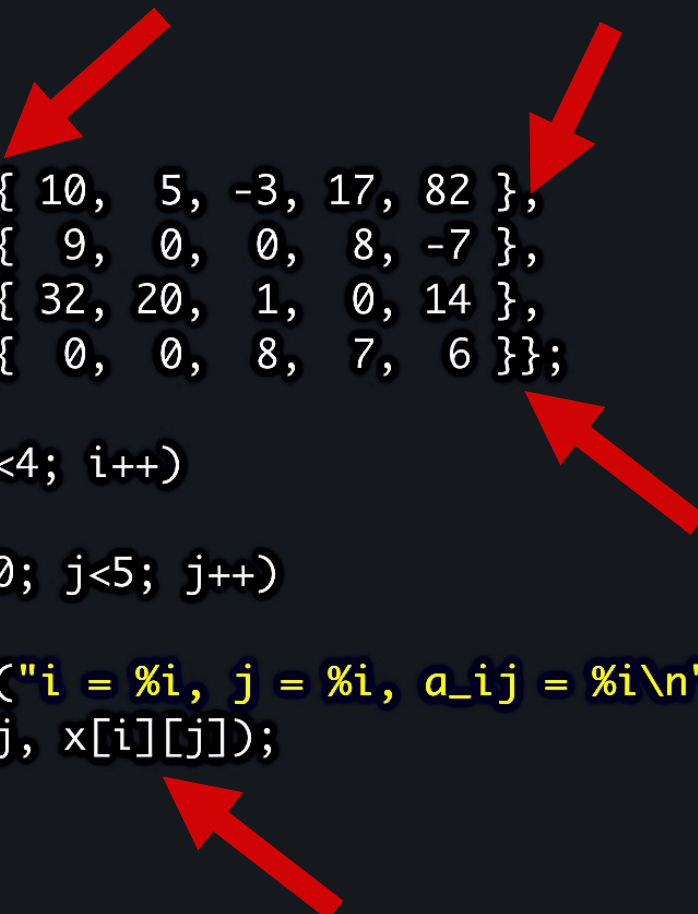
You can create n-dimensional arrays easily

```
#include <stdio.h>

int main (void)
{
    int x[4][5] = {{ 10,  5, -3, 17, 82 },
                   {  9,  0,  0,  8, -7 },
                   { 32, 20,  1,  0, 14 },
                   {  0,  0,  8,  7,  6 }};

    for (int i=0; i<4; i++)
    {
        for (int j=0; j<5; j++)
        {
            printf ("i = %i, j = %i, a_ij = %i\n",
                    i, j, x[i][j]);
        }
    }

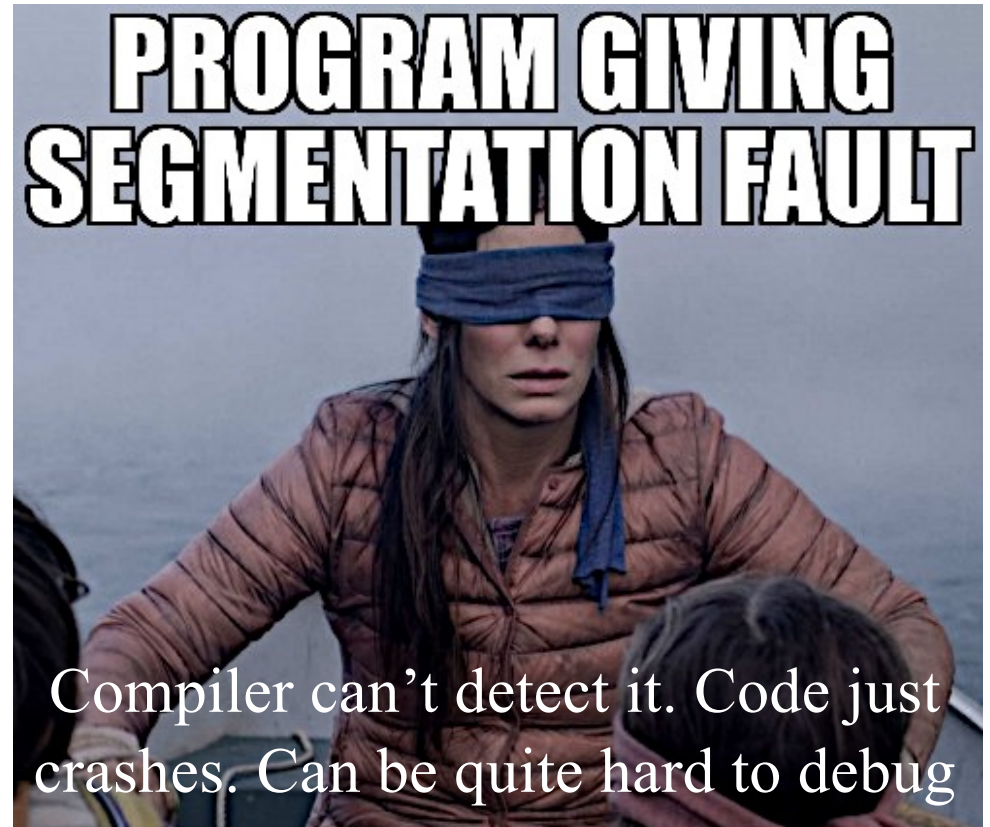
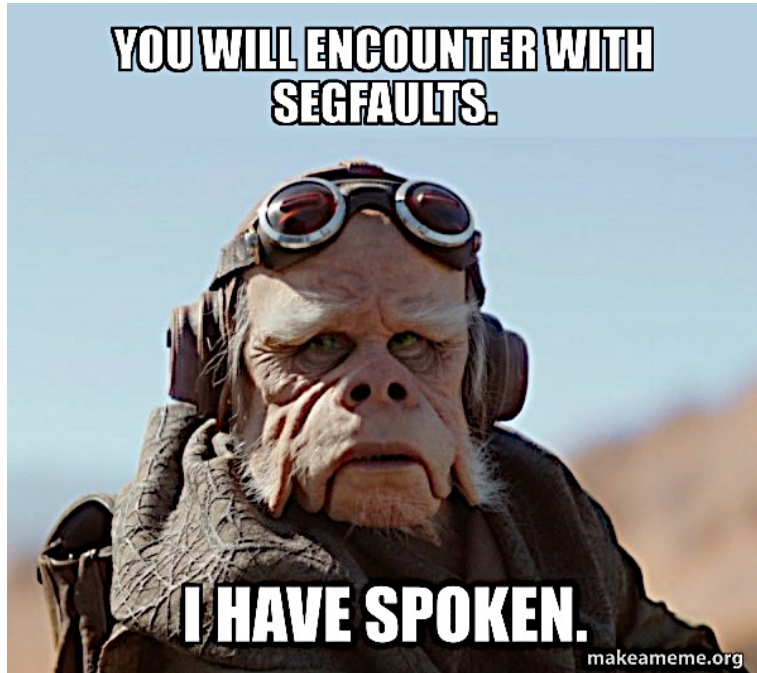
    return 0;
}
```



```
whovian@phy504:~/host$ ./array7
i = 0, j = 0, a_ij = 10
i = 0, j = 1, a_ij = 5
i = 0, j = 2, a_ij = -3
i = 0, j = 3, a_ij = 17
i = 0, j = 4, a_ij = 82
i = 1, j = 0, a_ij = 9
i = 1, j = 1, a_ij = 0
i = 1, j = 2, a_ij = 0
i = 1, j = 3, a_ij = 8
i = 1, j = 4, a_ij = -7
i = 2, j = 0, a_ij = 32
i = 2, j = 1, a_ij = 20
i = 2, j = 2, a_ij = 1
i = 2, j = 3, a_ij = 0
i = 2, j = 4, a_ij = 14
i = 3, j = 0, a_ij = 0
i = 3, j = 1, a_ij = 0
i = 3, j = 2, a_ij = 8
i = 3, j = 3, a_ij = 7
i = 3, j = 4, a_ij = 6
```

Arrays

You can write on array locations that have no memory allocation: **segfault**



- Ref. 1 from [codeforgeeks](#)
- Ref. 2 from [StackOverflow](#)
- Ref. 3 from [StackOverflow](#)

Arrays

Even worse than segfault is when C just return garbage (out of bounds)

```
#include <stdio.h>

int main (void)
{
    int x[5];

    x[0] = 197;
    x[1] = 200;
    x[2] = -100;
    x[3] = x[0] + x[1];
    x[4] = 350;

    for (int i=0; i<100; ++i)
    {
        printf ("x[%i] = %i\n", i, x[i]);
    }
    return 0;
}
```

out of bounds access to
array is **undefined**
behavior (it could cause
a segmentation fault, but
it doesn't have to)

```
whovian@phy504:~/hos
whovian@phy504:~/hos
x[0] = 197
x[1] = 200
x[2] = -100
x[3] = 397
x[4] = 350
x[5] = 64
x[6] = 946356992
x[7] = 244184714
x[8] = 0
x[9] = 0
x[10] = 25616515
x[11] = 64
x[12] = 107
x[13] = 1
x[14] = 25187512
x[15] = 64
x[16] = 27469728
x[17] = 1
x[18] = 4457
```

Arrays

Undefined behavior is the most difficult error to debug (the code just runs)



- Ref. 1 from [codeforgeeks](#)
- Ref. 2 from Prof. [John Regehr's Blog](#)

Functions in C

```
GNU nano 4.8
#include <stdio.h>

void swap(int a, int b)
{
    const int t = a;
    a = b;
    b = t;

    printf("swap: a = %d, b = %d\n", a, b);
}

int main ()
{
    const int x = 2;
    const int y = 3;

    swap (2, 3);

    return 0;
}
```

arguments

return type

void -> no return

whovian@phy504:~/host\$ gcc functions.c -o functions

whovian@phy504:~/host\$./functions

swap: a = 3, b = 2

whovian@phy504:~/host\$

Functions accept arguments, do something, and return values

Of course, this is C, and any basic definitions can be tweaked.

Functions may not return anything

Functions may also not accept arguments

Functions in C

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

```
void swap(int a, int b)
{
```

```
    const int t = a;
    a = b;
    b = t;
```

```
    printf("swap: a = %d, b = %d\n", a, b);
}
```

```
int main ()
{
```

```
    int x = 2;
    int y = 3;
```

```
    swap (x, y);
```

```
    printf("swap: a = %d, b = %d\n", x, y);
```

```
    return 0;
```

```
}
```

```
whovian@phy504:~/host$ gcc functions2.c -o functions2
```

```
whovian@phy504:~/host$ ./functions2
```

```
swap: a = 3, b = 2
```

```
swap: a = 2, b = 3
```

x and y are copied

x and y are copied

Function copy arguments (**pass by value**). You can't change the parameter values via arguments*

*Unless you use points (next class)

Functions in C

Functions may not have any arguments

```
#include <stdio.h>

void PrintMessage (void)
{
    printf ("Programming is fun. But this"
           " function is boring.\n");
}

int main ()
{
    PrintMessage();

    return 0;
}
```

```
whovian@phy504:~/host$ gcc functions3.c -o functions3
whovian@phy504:~/host$ ./functions3
Programming is fun. But this function is boring.
whovian@phy504:~/host$
```

Functions in C

Function return values (just one!)

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

```
double KmhToMph (const double kmh)
```

```
{  
    return kmh * 0.62137119223;  
}
```

```
int main ()  
{  
    printf("1 km/h corresponds to "  
           "%lf miles/h \n", KmhToMph(1));  
  
    return 0;  
}
```

Having “magic numbers” out of thin air is bad coding practice.

Btw: not enough digits to maintain double precision

```
whovian@phy504:~/host$ gcc functions4.c -o functions4  
whovian@phy504:~/host$ ./functions4  
1 km/h corresponds to 0.621371 miles/h
```

Functions in C

mildly dangerous

(but important case): arrays

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

```
int min (int vals[10])
{
    int tmp = vals[0];

    for (int i=1; i<10; ++i)
    {
        if ( vals[i] < tmp )
        {
            tmp = vals[i];
        }
    }
    return tmp;
}
```

```
int main (void)
{
    int scores[10] = {2, 1, 3, 4, 5,
                     6, 9, 10, 7, 8};
    printf ("Minimum score is %i\n", min(scores));

    int scores2[5] = {2, 1, 3, 4, 5};
    printf ("Minimum score is %i\n", min(scores2));

    int scores3[11] = {2, 1, 3, 4, 5,
                      6, 9, 10, 7, 8, 0};
    printf ("Minimum score is %i\n", min(scores3));

    return 0;
}
```

```
whovian@phy504:~/host$ ./functions5
```

```
Minimum score is 1
```

```
Minimum score is 0
```

```
Minimum score is 1
```



wrong



wrong

UNDEFINED BEHAVIOUR

May be right/wrong – who knows? Nobody! Undefined

Wrong – but behavior is predictable

Functions in C

With VLAs and an extra integer argument, we can fix the problem

```
#include <stdio.h>

int min (int n, int vals[])
{
    if (n <= 0) ← Check input
    {
        printf ("Error, size = %i \n", n);
    }

    int tmp = vals[0];

    for (int i=1; i<n; ++i)
    {
        if ( vals[i] < tmp )
        {
            tmp = vals[i];
        }
    }
    return tmp;
}

int main (void)
{
    int scores[10] = {2, 1, 3, 4, 5,
                     6, 9, 10, 7, 8};
    printf ("Minimum score is %i\n", min(10, scores));

    int scores2[5] = {2, 1, 3, 4, 5};
    printf ("Minimum score is %i\n", min(5, scores2));

    int scores3[11] = {2, 1, 3, 4, 5,
                      6, 9, 10, 7, 8, 0};
    printf ("Minimum score is %i\n", min(11, scores3));

    return 0;
}
```

```
whovian@phy504:~/host$ ./functions5
Minimum score is 1
Minimum score is 1
Minimum score is 0
```



Functions in C

With arrays we CAN change the values of the function input

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

```
int min (int vals[10])  
{
```

```
    int tmp = vals[0];
```

```
    for (int i=1; i<10; ++i)  
    {
```

```
        if ( vals[i] < tmp )  
        {
```

```
            tmp = vals[i];
```

```
        }
```

```
    }  
    vals[1] = -1;  
    return tmp;  
}
```

```
int main (void)  
{
```

```
    int scores[10] = {2, 1, 3, 4, 5,  
                     6, 9, 10, 7, 8};
```

```
    printf ("Minimum score is %i\n", min(scores));
```

```
    printf ("scores[1] = %i \n", scores[1]);
```

```
    return 0;
```

```
}
```

```
whovian@phy504:~/host$ ./functions7
```

```
Minimum score is 1
```

```
scores[1] = -1
```

Why? Because under the hood arrays are basically pointers!
(see next class)

Functions in C

With arrays we CAN change the values of the function input

```
#include <stdio.h>

int min (const int vals[10])
{
    int tmp = vals[0];

    for (int i=1; i<10; ++i)
    {
        if ( vals[i] < tmp )
        {
            tmp = vals[i];
        }
    }

    vals[1] = -1;
    return tmp;
}
```

```
whovian@phy504:~/host$ gcc functions8.c -o functions8
```

```
functions8.c: In function 'min':
```

```
functions8.c:14:13: error: assignment of read-only location '*(vals + 4)'
```

```
14 |     vals[1] = -1;
```

```
|
```

```
^
```

```
int main (void)
{
    int scores[10] = {2, 1, 3, 4, 5,
                      6, 9, 10, 7, 8};
    printf ("Minimum score is %i\n", min(scores));
    printf ("scores[1] = %i \n", scores[1]);

    return 0;
}
```

If your function does not change array values, make the argument constant!

It also helps the compiler

Homework I

The website (<https://lucidar.me/en/c-class/lesson-08-08-random-numbers-in-c/>) has a nice and simple explanation for creating random numbers in C.

- Write a program that simulates a dice with 18 faces
- Write a program that computes 10 uniform random numbers between $[0,1]$, then apply the [Box-Muller transformation](#) to create 10 numbers that obey the normal distribution

Hint: These websites teach how to compute sin, cos, and ln in C

- https://www.tutorialspoint.com/c_standard_library/c_function_sin.htm
- https://www.tutorialspoint.com/c_standard_library/c_function_cos.htm
- https://www.tutorialspoint.com/c_standard_library/c_function_log.htm

Homework II

- Write a C program that finds the greatest common divisor of two nonnegative integer values.

Hint: Below is the pseudocode that describes the algorithm

```
Algorithm GCD(a, b):  
    Input: two nonnegative integers a and b  
    Output: greatest common divisor of a and b  
  
    while b ≠ 0 do  
        temp ← b  
        b ← a mod b  
        a ← temp  
    end while  
  
    return a
```

Homework III

- Write a C program that reverses the digits of an integer.
Hint: Below is the pseudocode that describes the algorithm

```
Algorithm ReverseDigits(num):  
  Input: integer num  
  Output: integer reversedNum (the digits of num in reverse order)  
  
  // Determine the sign of the number and work with its absolute value.  
  if num < 0 then  
    sign <- -1  
    num <- absolute(num)  
  else  
    sign <- 1  
  
  reversedNum <- 0  
  
  // Extract digits from num and build the reversed number.  
  while num ≠ 0 do  
    digit <- num mod 10          // Get the last digit.  
    reversedNum <- reversedNum * 10 + digit // Append the digit.  
    num <- num // 10             // Remove the last digit (integer division).  
  end while  
  
  // Restore the original sign.  
  reversedNum <- reversedNum * sign  
  
  return reversedNum
```

Homework IV

- Write a C program to generate and display a table of n and n^2 , for integer values of n ranging from 1 to 10. Be certain to print appropriate column headings
- Write a C program that asks the user for an expression and computes the result. Valid options: $\text{val1} + \text{val2}$; $\text{val1} - \text{val2}$; $\text{val1} * \text{val2}$ or $\text{val1} / \text{val3}$;
- Write a program that computes the numerical integration via trapezoidal rule of a function (e.g., $f(x) = x^2$ from 0 to 10).
- Write a program in C to sort elements of the array in ascending order.
- Write a program in C to print all unique elements in an array.

Additional Reading

- Additional reading on C variable initialization:
<https://stackoverflow.com/a/6212973/2472169>
- *Geek for Geeks* (GfG) is a great website to learn data structures and interesting features in C/C++. Here is the GfG discussion about stacks <https://origin.geeksforgeeks.org/introduction-to-stack-data-structure-and-algorithm-tutorials/>

Additional Reading: Do not smash the stack!



Site Search

.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r00t, and Underground.Org
bring you

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

by Aleph One
aleph1@underground.org

`smash the stack` [C programming] n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind. Variants include trash the stack, scribble the stack, mangle the stack; the term mung the stack is not used, as this is never done intentionally. See spam; see also alias bug, fandango on core, memory leak, precedence lossage, overrun screw.

This is much more
dangerous than you
think

The website

<https://insecure.org/sf/smashstack.html>

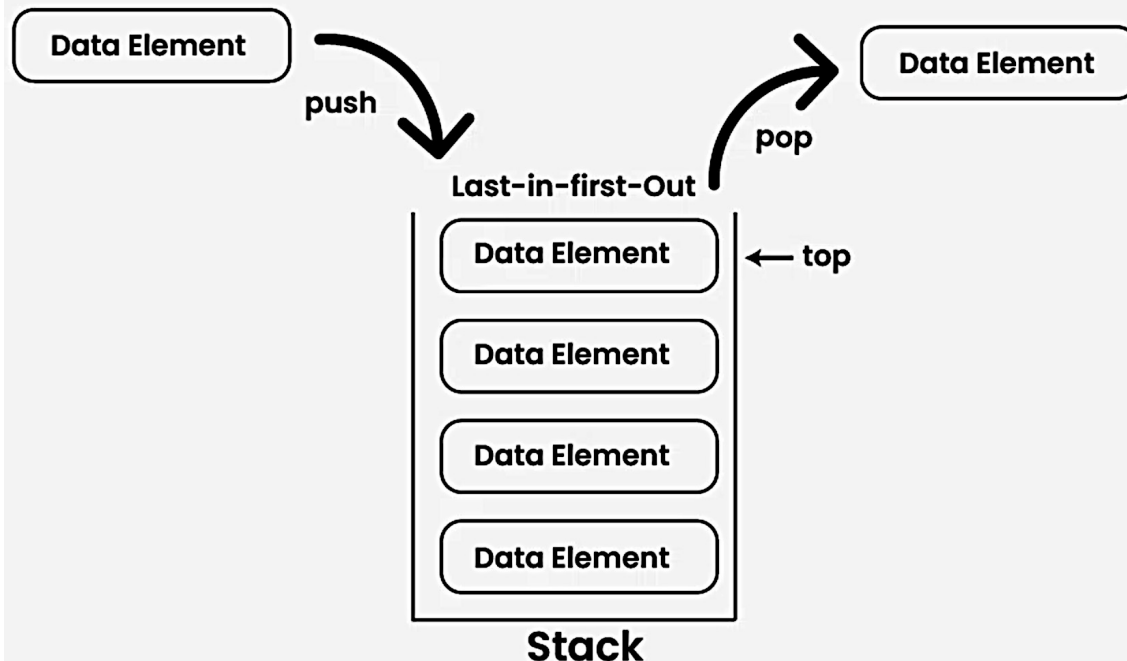
shows how this type
of bug can be
exploited to take
control of a system!

Additional Reading: What is a stack?

An abstract linear data type (we will implement stack w/ linked-lists)

Stack obeys the LIFO (Last In, First Out) Principle; the last element inserted is the first to be popped out.

Representation of Stack Data Structure



Operations allowed

- **push()** to insert an element into the stack
- **pop()** to remove an element from the stack
- **top()** Returns the top element of the stack.
- **isEmpty()** returns true if stack is empty else false.
- **isFull()** returns true if the stack is full else false.

Additional Reading: Do Braces actually define stack?

Technically, no. But for all practical purposes, assume it does otherwise you may deal with undefined behavior



46

The time during which the variable is *actually* taking up memory is obviously compiler-dependent (and many compilers don't adjust the stack pointer when inner blocks are entered and exited within functions).



However, a closely related but possibly more interesting question is whether the program is allowed to access that inner object outside the inner scope (but within the containing function), ie:



```
void foo() {  
    int c[100];  
    int *p;  
  
    {  
        int d[200];  
        p = d;  
    }  
  
    /* Can I access p[0] here? */  
  
    return;  
}
```

<https://stackoverflow.com/a/2762213/2472169>

(In other words: is the compiler *allowed* to deallocate `d`, even if in practice most don't?).

The answer is that the compiler *is* allowed to deallocate `d`, and accessing `p[0]` where the comment indicates is undefined behaviour (the program is *not* allowed to access the inner object outside of the inner scope). The relevant part of the C standard is 6.2.4p5:

Additional Reading: Do not get too comfortable in C

Perfect C code can be hard to read: obfuscated C code.

The following code won a competition on the topic (2020).

The International Obfuscated C Code Contest

```
GNU nano 8.1                                     prog.c
int main(int b,char**i){
    long long n=B;
    long long a=I^n;
    long long r=(a/b&a)>>4;
    long long y=atoi(*++i);
    long long _=((a^n/b)*(y>>T)|y>>S)&r)|(a^r);
    printf("%.8s\n",(char*)&_);
    return 0;
}
```

Perfect legal C code: only uses basic C operations

Additional Reading: Do not get too comfortable in C

Perfect C code can be hard to read: obfuscated C code.

The following code won a competition on the topic (2020).

[The International Obfuscated C Code Contest](#)

```
RiverSong:Lecture9 vivianmiranda$ clang -include stdio.h -include stdlib.h -Wall -Weverything
-pedantic -DB=6945503773712347754LL -DI=5859838231191962459LL -DT=0 -DS=7 -o prog prog.c
warning: include location '/usr/local/include' is unsafe for cross-compilation [-Wpoison-syste
m-directories]
1 warning generated.
RiverSong:Lecture9 vivianmiranda$ ./prog 1
00000001
RiverSong:Lecture9 vivianmiranda$ clang -include stdio.h -include stdlib.h -Wall -Weverything
-pedantic -DB=6945503773712347754LL -DI=5859838231191962459LL -DT=0 -DS=7 -o prog prog.c
warning: include location '/usr/local/include' is unsafe for cross-compilation [-Wpoison-syste
m-directories]
1 warning generated.
RiverSong:Lecture9 vivianmiranda$ ./prog_be 1
10000000
```

Perfect legal C code: only uses basic C operations