Week 11 Lecture 1

Deep C Secrets: *Interesting aspects of programming in C*

Deep C Secrets

- This book will take your programming to the level after the next level.
- But even for the novice programmer it has some interesting and understandable tidbits.
- Should definitely be in your library.

Expert C Programming : Deep C Secrets

Book by Peter van der Linden

4.3/5 · Goodreads

This book is for the knowledgeable C programmer, this is a second book that gives the C programmers advanced tips and tricks. This book will help the C programmer reach new heights as a professional. ... Google Books

Originally published: 1994

Author: Peter van der Linden

Software Dogma

The Switch Statement that defeated AT&T

The Switch Statement

- The `switch` statement tests the value of a variable and compares it with multiple **cases**.
- When a match is found, a block of statements associated with that case is executed.
- When a `break` statement is reached, the `switch` terminates, and the *flow of control* jumps to the next line following the `switch` statement.
- If no `break` appears, the *flow of control* **will fall through** to subsequent cases until a `break` is reached.

```
switch ( expression ) {

   case constant-expression:
      statement(s);
      break;                        /* Optional */
   case constant-expression:
      statement(s);
      break;                        /* Optional */

   /* you can have any number of case
      statements */

   default:                         /* Optional */
      statement(s);
      break;                        /* Optional */
}
```

Other Notes about `switch`

- The `break` statement is optional and if omitted then execution will continue on to the next case.
  - The flow of control will **fall through** to the following cases until a `break` is reached. If this is not what was intended (and it probably was not), then bad things will probably happen.
- You can of course nest `switch` statements - `switch` statements inside other `switch` statements. Nested `switch` statement can easily and quickly become **unreadable**.

```
i = 2;
switch (i) {
   case 1:  printf ("case 1\n");
   case 2:  printf ("case 2\n");
   case 3:  printf ("case 3\n");
   case 4:  printf ("case 4\n");
   default: printf ("default\n");
}
```

This is known as "**fall through**" and was intended to allow **common** end processing to be done, after some **case-specific** preparation had occurred.

In practice it's a severe **misfeature**, as almost all case actions end with a `break;`.

This is a replica of the code that caused a major disruption of AT&T phone service throughout the U.S. AT&T's network was in large part unusable for about nine hours starting on the afternoon of January 15, 1990.

```
network code()
{
  switch (line) {
      case THING1:
        doit1();
        break;
      case THING2:
        if (x == STUFF) {
            do_first_stuff();
            if (y == OTHER_STUFF)
                break;
                do_later_stuff();
            }         /* Coder meant to break to here... */
            initialize_modes_pointer();
            break;
      default:
         processing();
  }                   /* But actually broke to here! */
  use_modes_pointer();/*   leaving the modes_pointer */
                      /*               uninitialized */
}
```

All because of a `switch` statement…

- The programmer wanted to break out of the `if` statement but

  - `break` gets you out of the nearest *enclosing iteration or switch statement.*

  - In this code it broke out of the switch, and executed the call to `use_modes_pointer()` but the necessary initialization had not been done, causing a failure further on.

All because of a `switch` statement…

- This code eventually caused the first major network problem in AT&T's 114-year history.

- The supposedly fail-safe design of the network signalling system actually spread the fault in a chain reaction, bringing down the entire long distance network…and it all rested on a C **`switch`** statement!

rewrite1_att.c

```
switch (line) {
   case THING1:
     printf ( "doit1();\n" );
     break;
   case THING2:
     if (x == STUFF) {
        printf ( "do_first_stuff();\n" );
        if (y == OTHER_STUFF)
           break;
        printf ( "do_later_stuff();\n" );
     }
     printf ( "initialize_modes_pointer();\n" );
     initial = 1;
     break;
   default:
     printf ( "processing();\n" );
}
printf ( “use_modes_pointer();\n");
printf ( "initial = %d\n", initial);
```

rewrite2_att.c

```
 switch (line) {
    case THING1:
      printf ( "doit1();\n" );
      break;
    case THING2:
      if (x == STUFF) {
          printf ( "do_first_stuff();\n" );
          if (y != OTHER_STUFF) {
             printf ( "do_later_stuff();\n" );
          }
      }
      printf ( "initialize_modes_pointer();\n" );
      initial = 1;
      break;
    default:
      printf ( "processing();\n" );
      break;
}
printf ( "use_modes_pointer();\n" );
printf ( "initial = %d\n", initial );
```

```
$ ./att 1 10 20
doit1();
use_modes_pointer();
initial = 0

$ ./att 2 10 20
do_first_stuff();
use_modes_pointer();
initial = 0

$ ./att 2 20 20
initialize_modes_pointer();
use_modes_pointer();
initial = 1
```

```
$ ./rewrite1_att 1 10 20
doit1();
use_modes_pointer();
initial = 0

$ ./rewrite1_att 2 10 20
do_first_stuff();
use_modes_pointer();
initial = 0

$ ./rewrite1_att 2 20 20
do_first_stuff();
use_modes_pointer();
initial = 0
```

```
$ ./rewrite2_att 2 10 20
do_first_stuff();
initialize_modes_pointer();
use_modes_pointer();
initial = 1
```

```
$ ./rewrite2_att 2 20 20
initialize_modes_pointer();
use_modes_pointer();
initial = 1
```

Handy Heuristic

Making String Comparison Look More ***Natural***

The Problem with `strcmp()`

- One of the problems with the `strcmp()` routine to compare two strings is that it returns **zero** if the strings are identical.

- This leads to convoluted code when the comparison is part of a conditional statement:

```
if (!strcmp(s, ”volatile”)) return QUAL;
```

- A zero result indicates **false**, so we have to *negate* it to get what we want.

Re-Define `strcmp()`

- Use a definition so that the code expresses what is happening in a more natural style.
- Set up the definition:

```
#define STRCMP(a,R,b) (strcmp(a,b) R 0)
```

- Now you can write a string in the natural style

```
if ( STRCMP ( s, ==, “volatile” ) ) …
```

Can we do better than the Deep C?

```
int strequal ( char *stringA, char *stringB );
```

- This function returns 1 if the strings are the same and 0 when they are not so that you can put the following in your code:

```
if ( strequal ( argv[1], argv[2] ) ) {

      printf ( "Equal\n" );

} else {

      printf ( "Not equal\n" );

}
```

Overloading

Why use two symbols when one will do?

Overloading *

```
p = N * sizeof * q;
```

- Quickly now, are there **two** multiplications or only **one**?
  - The answer is that there's only **one** multiplication.
- `sizeof` is an operator that takes as its operand the thing pointed to by `q`, in other words `*q`.
- When `sizeof` 's operand is a **type** it has to be enclosed in parentheses, but for a **variable** this is not required.

A little more complicated…

- `apple = sizeof (int) * p;`
- What does this mean?
  - Is it the size of an `int`, multiplied by `p`?
  - Or the size of whatever `p` points at cast to an `int`?
  - ??

```c
int apple = 0;
int p = 2;
float p2 = 2.5;
int *ptr = &p;

apple = sizeof (int);
printf ( "Apple = %d\n", apple );

apple = sizeof (int) * p;
printf ( "Apple = %d\n", apple );

apple = sizeof (int) * p2;
printf ( "Apple = %d\n", apple );

apple = sizeof (* ptr);
printf ( "Apple = %d\n", apple );
```

```
$ ./overload1
Apple = 4
Apple = 8
Apple = 10
Apple = 4
```

No Space -
Take a Guess

?? What The ???

Spaces Do Make a Difference

- What do you think the following code means?

```
z = y+++x;
```

- Does it mean?

```
z = y + ++x;

z = y++ + x;
```

Maximal Munch Strategy

- The ANSI standard specifies a convention that has come to be known as the **maximal munch** strategy.

- **Maximal munch** says that if there's more than one possibility for the next token, the compiler will prefer to bite off the one involving the longest sequence of characters.

```
z=y+++x
```

will be parsed as

```
z = y++ + x
```

Munch,Munch, Munch

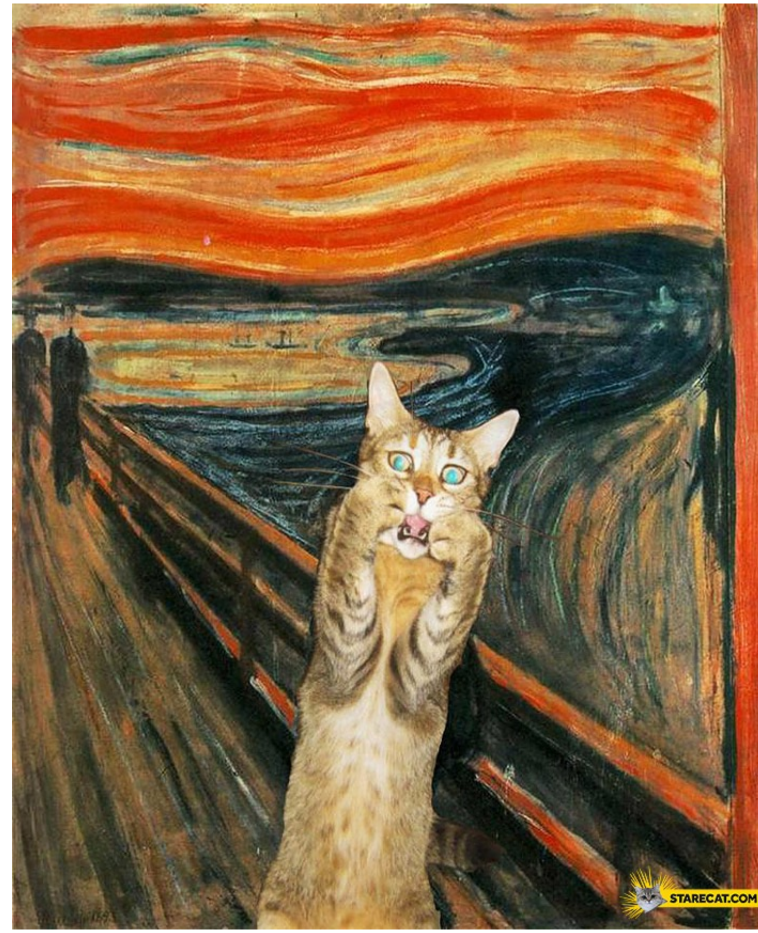
- But what about

```
z = y+++++x;
```

- Maximum munch will generate:

```
z = y++ ++ + x;
```

- **But this is an error!**




```
$ gcc -ansi -Wall noSpaces.c -o noSpaces
noSpaces.c: In function ‘main’:
noSpaces.c:9:11: error: lvalue required as increment operand
   z = y+++++x;
          ^~
```

Munch,Munch, Munch

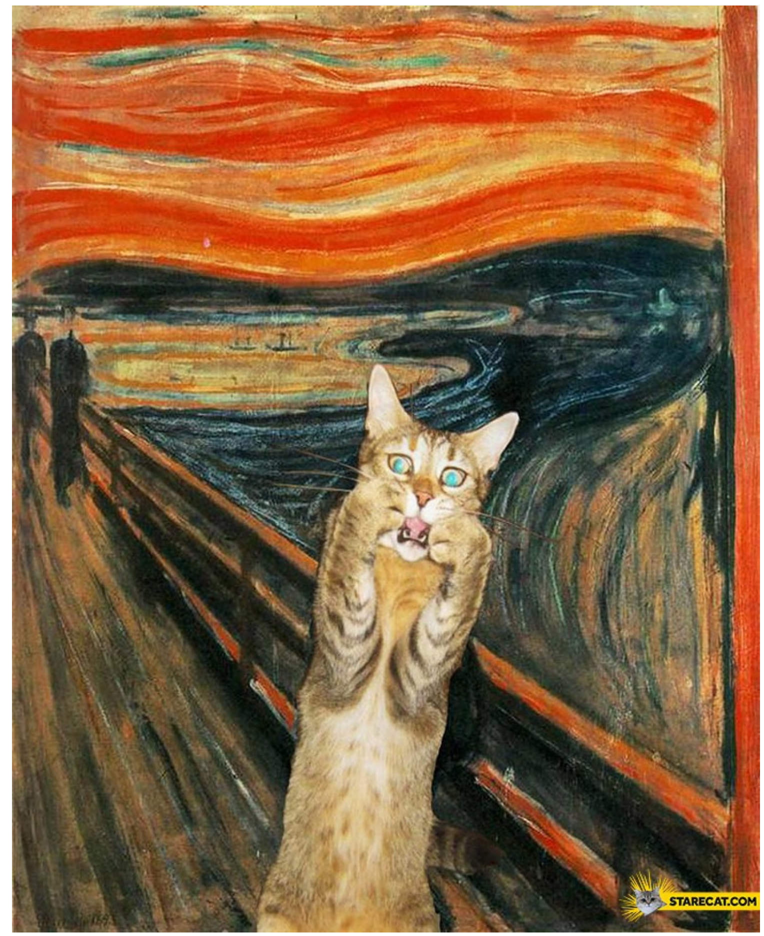
- Is there a valid interpretation for `z = y+++++x;` ?

```
int x = 10;
int y = 20;
int z = 0;

z = y++ + ++x;

printf ( "%d %d %d\n",x,y,z );
```

$ ./spaces
11 21 31



And now a few things about size…

Size does matter!

Integers

TYPE	SIZE	MIN	MAX
int	4	-2147483648	2147483647
unsigned int	4	0	4294967295
long	8	-9223372036854775808	9223372036854775807
unsigned long	8	0	18446744073709551615

limits.h

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

int main ()
{
   printf ( "TYPE  SIZE    MIN    MAX\n" );
   printf ( "int   %lu     %d     %d\n",
      sizeof(int), INT_MIN, INT_MAX );
   printf ( "unsigned int  %lu    0   %u\n",
      sizeof(unsigned int), UINT_MAX );
   printf ( "long          %lu    %ld %ld\n",
      sizeof(long), LONG_MIN, LONG_MAX );
   printf ( "unsigned long %lu    0   %lu \n",
      sizeof(unsigned long), ULONG_MAX );
}
```

```
$ ./limits
TYPE             SIZE    MIN                     MAX
int              4       -2147483648              2147483647
unsigned int     4       0                       4294967295
long             8       -9223372036854775808     9223372036854775807
unsigned long    8       0                       18446744073709551615
```

Integers

- The actual size of integer types varies by implementation.
- The standard only requires size relations between the data types and minimum sizes for each data type.
- The relation requirements are that `long` is not smaller than `int`, which is not smaller than `short`.
- `char`'s size is always the minimum supported data type.
- The minimum size for `char` is 8 bits, the minimum size for `int` is 16 bits, for `long` it is 32 bits and `long long` must contain at least 64 bits.

Floating Point

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

int main ()
{
   printf ( "TYPE SIZE  MIN      MAX\n" );
   printf ("float %lu   %e %e\n", sizeof(float),
      FLT_MIN, FLT_MAX );
   printf ("double   %lu   %e %e\n", sizeof(double),
      DBL_MIN, DBL_MAX );
}
```

```
TYPE    SIZE   MIN           MAX
float   4      1.175494e-38  3.402823e+38
double  8      2.225074e-308 1.797693e+308
```

Overflow

There is more than one way to solve a problem.

Overflow

```
#include <stdio.h>
#include <limits.h>
int main ()
{
   int firstInt, secondInt;
   int leftInt, midInt, rightInt;

   firstInt = INT_MAX;
   secondInt = firstInt + 1;
   printf ( "%d + 1 = %d\n", firstInt, secondInt );
   firstInt = INT_MIN;
   secondInt = firstInt - 1;
   printf ( "%d - 1 = %d\n", firstInt, secondInt );
```

2147483647 + 1 = -2147483648
-2147483648 - 1 = 2147483647

```c
/*
 * Find the midpoint between firstInt and secondInt - First Try
 */

leftInt = 100;
rightInt = INT_MAX - 10;

midInt = ( leftInt + rightInt ) / 2;

printf ( "Midpoint between %d and %d is %d\n",
   leftInt, rightInt, midInt );

/* Second Try */

midInt = ( ( rightInt - leftInt ) / 2 ) + leftInt;

printf ( "Midpoint between %d and %d is %d\n",
   leftInt, rightInt, midInt );
```

```
Midpoint between 100 and 2147483637 is -1073741779
Midpoint between 100 and 2147483637 is 1073741868
```