Common Errors in C

David Chisnall

February 15, 2011

The C Preprocessor

- Runs before parsing
- Allows some metaprogramming

Preprocessor Macros Are Not Functions

- The preprocessor performs token substitution
- It is completely unaware of the language semantics

```
#define ANSWER 42
test(ANSWER); // replaced by test(42)
#define CALL(x) (x)()
CALL(y()); // replaced by (y())()
```

Example Failure

```
#define MIN(x, y) ((x<y) ? x : y)
```

- Looks correct?
- What happens if the arguments are side effects?

Side Effects Evaluated Twice

```
int min = MIN(a(), b());
// Expands to:
int min = ((a()<b()) ? a() : b());
```

- Whichever function has the lower value is called twice
- If it has side effects, this is very bad
- If it doesn't, it's still overhead

Solution: Use Functions?

```
int min(int x, int y)
{
    if (x<y)
    {
        return x;
    }
    return y;
}
...
min(a(), b());
```

- Looks safe - functions are called then the result passed as arguments.
- What happens if `a()` and `b()` return `float`s?

Solution 2: More Complex Macro

```
#define MIN(x, y) ({ \
    __typeof__(x) _x = x;\
    __typeof__(y) _y = y;\
    _x < _y ? _x : y;\
    })
```

- Works correctly
- But uses two GCC-specific extensions
- It is not possible to portably solve this problem in C99!

Type Promotion

- Arithmetic results in C depend on the types of the operands
- This may not do what you want!

```
int64_t multiply_extend(int32_t a, int32_t b)
{
    // Wrong!
    int64_t result = a*b;
    // Correct:
    result = (int64_t)a*b;
    return result;
}
```

The Rules (Simplified)

- Arithmetic operations on the same type evaluate to that type
- Arithmetic operations on different types use the one with the wider range
- The actual rules are more complex than this!
- If you're confused, explicitly cast both arguments.
- Overly verbose code is better than buggy code

```
double half = 1/2; // Evaluates to 0!
```

Assignment Instead of Comparison

```
if (a = b)
```

- What does this do?
- Assigns `b` to `a`
- Tests if `a` is zero
- Enters the `if` block if it is non-zero

Avoiding Accidental Assignment

- Put rvalue on the left side of comparions (e.g. `0 == a` not `a == 0`)
- Explicitly add `0 ==` when you want to compare against 0.
- Turn on compiler warnings...
- ...and *pay attention to them!*

Forgetting How Switch Statements Work

```
switch (expression)
{
     case 1:
          doSomething();
     case 2:
          doSomethingElse();
     default:
          giveUp();
}
```

- Looks right?
- `giveUp()` is called in all cases
- `doSomethingElse()` is called even for values of 1
- Sometimes you want this, but usually you don't

Forgetting How Switch Statements Work

```
switch (expression)
{
     case 1:
          doSomething();
          break;
     case 2:
          doSomethingElse();
          break;
     default:
          giveUp();
}
```

- This version is right
- Note for C# programmers: C# switch statement works differently!

Comparing Strings with ==

```
if ("test" == a)
```

- Why is this wrong?
- C does not have real strings
- It uses pointers to characters / character arrays instead
- This compares two pointers - they are only the same if they point to the same string, not two identical strings in different places in memory

Correct String Comparison

```
if ((NULL != a) && (strcmp("test", a) == 0))
```

- `NULL` test is required because `strcmp()` expects valid pointers
- The `strcmp()` function returns an ordering, so you can compare against zero to sort strings

Arrays Are Pointers - Mostly

```
int example(void)
{
    char *a = "foo";
    char b[] = "foo";
    ...
```

- Are these two the same?
- *"foo"* is a global constant string
- `a` is a pointer to that string
- `b` is a copy of that string on the stack

Printf and Scanf Problems

```
int a = 12;
printf("Current value: %f, old value: %d\n", a);
scanf("%d", a);
```

- Three bugs here, what are they?
- `a` is passed as the first argument to `printf`, but the format specifier tells it to expect a floating point value.
- One of the arguments to `printf` is missing
- `scanf` reads values, so expects pointer arguments
- These functions expect variable arguments, so the compiler needs special logic to check them.
- Most compilers will check arguments to these functions, don't ignore the warnings!

More Variadic Function Problems

```
expects_null_terminated(a, b, c, d, e, 0);
```

- What's wrong here?
- Hint: It will work correctly on *most* platforms.
- 0 is passed as an `int`
- On some platforms, `int` is 4 bytes, pointers are 8 bytes
- The callee will read 8 bytes, test if they're `NULL` and treat them as a pointer if they're not.
- Bigger problem in C++ because standard C++ has no proper `NULL` (fixed in C++0X)

Not Understanding Strings

- Most languages have a proper string type
- C uses character (byte) arrays
- No embedded length, end is identified by a 0 byte

Common Errors With Strings

```
int len = strlen(str);
char buffer[len];
strncpy(buffer, str, len);
```

- Allocates enough space for the characters in the string
- But not enough for the NULL terminator

Don't Forget the Terminator

```
int len = strlen(str);
char buffer[len+1];
memcpy(buffer, str, len);
buffer[len] = '\0';
```

Unsafe String Functions

```
strcpy(buffer, input);
```

- What happens if `buffer` is smaller than `input`?
- Memory corruption!

Slightly Less Unsafe String Functions

```
strncpy(buffer, input, length);
```

- What happens if `buffer` is smaller than `input`?
- Silent truncation, `buffer` is not terminated

Safe String Functions

```
int len = 128;
char buffer[len];
if (len < strlcpy(buffer, input, len))
{
// Handle case where truncation would occur
}
```

- Output string is *always* terminated.
- Return value from `strlcpy()` is the size of `input`
- Easy to use safely
- Not in glibc, so unavailable on GNU/Linux

Using gets

```
// Should be big enough
char *buffer = malloc(1024);
// Ooops, this function has no way of knowing
   how big the buffer is
gets(buffer);
```

- It is impossible to use `gets()` correctly
- It is deprecated in C99
- It is removed in C1X
- If someone suggests using it, ignore anything else they say about anything.

Questions?