SectorC: A C Compiler in 512 bytes

SectorC (github) is a C compiler written in x86-16 assembly that fits within the 512 byte boot sector of an x86 machine. It supports a subset of C that is large enough to write real and interesting programs. It is quite likely the smallest C compiler ever written.

In a base64 encoding, it looks like this:

```
6gUAwAdoADAfaAAgBzH/6DABPfQYdQXoJQHr8+gjAVOJP+gSALDDqluB+9lQdeAG/zdoAEAfy+gI
AegFAYnYg/hNdFuE9nQNsOiqiwcp+IPoAqvr4j3/FXUG6OUAquvXPVgYdQXoJgDrGj0C2nUGV+gb
AOsF6CgA68Ow6apYKfiD6AKrifgp8CaJRP7rrOg4ALiFwKu4D4Srq1fonP9ewz2N/HUV6JoA6BkA
ieu4iQRQuIs26IAAWKvD6AcAieu4iQbrc4nd6HkA6HYA6DgAHg4fvq8Bra052HQGhcB19h/DrVCw
UKroWQDoGwC4WZGrW4D/wHUMuDnIq7i4AKu4AA+ridirH8M9jfx1COgzALiLBOucg/j4dQXorf/r
JIP49nUI6BwAuI0G6wyE0nQFsLiq6wa4iwarAduJ2KvrA+gAAOhLADwgfvkx2zHJPDkPnsI8IH4S
weEIiMFr2wqD6DABw+gqAOvqicg9Ly90Dj0qL3QSPSkoD5TGidjD6BAAPAp1+eu86Ln/g/jDdfjr
slIx9osEMQQ8O3QUuAACMdLNFIDkgHX0PDt1BIkEMcBaw/v/A8H9/yvB+v/34fb/I8FMAAvBLgAz
wYQA0+CaANP4jwCUwHf/lcAMAJzADgCfwIUAnsCZAJ3AAAAAAAAAAAAAAAAAAAAAAAAAAAAAVao=
```

Supported language

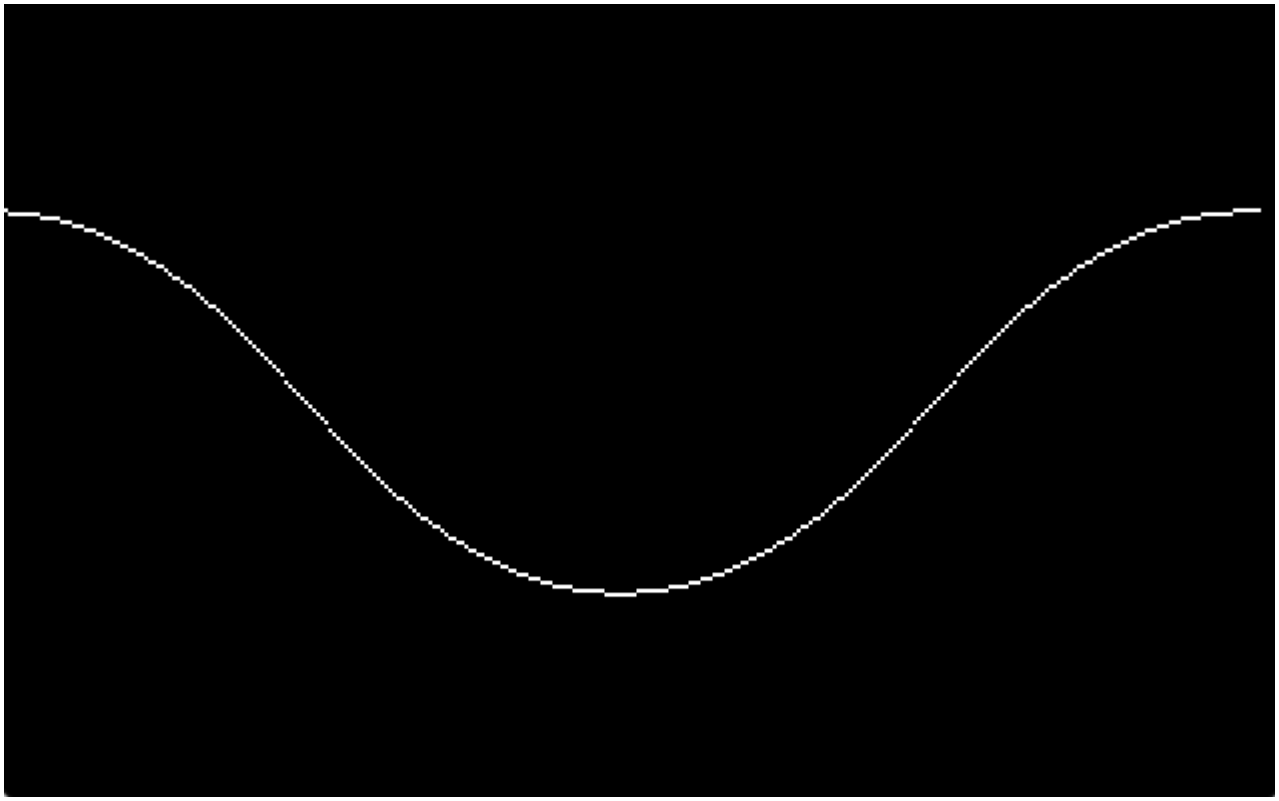
A fairly large subset is supported: global variables, functions, if statements, while statements, lots of operators, pointer dereference, inline machine-code, comments, etc. All of these features make it quite capable.

For example, the following program animates a moving sine-wave:

```
int y;
int x;
int x_0;
void sin_positive_approx()
{
  y = ( x_0 * ( 157 - x_0 ) ) >> 7;
}
void sin()
{
  x_0 = x;
  while( x_0 > 314 ){
    x_0 = x_0 - 314;
  }
  if( x_0 <= 157 ){
    sin_positive_approx();
  }
  if( x_0 > 157 ){
    x_0 = x_0 - 157;
    sin_positive_approx();
    y = 0 - y;
  }
  y = 100 + y;
}
```

```
int offset;
int x_end;
void draw_sine_wave()
{
  x = offset;
  x_end = x + 314;
  while( x <= x_end ){
    sin();
    pixel_x = x - offset;
    pixel_y = y;
    vga_set_pixel();
    x = x + 1;
  }
}

int v_1;
int v_2;
void delay()
{
  v_1 = 0;
  while( v_1 < 50 ){
    v_2 = 0;
    while( v_2 < 10000 ){
      v_2 = v_2 + 1;
    }
    v_1 = v_1 + 1;
  }
}

void main()
{
  vga_init();

  offset = 0;
  while( 1 ){
    vga_clear();
    draw_sine_wave();

    delay();
    offset = offset + 1;
    if( offset >= 314 ){ // mod the value to avoid 2^16 integer overflow
      offset = offset - 314;
    }
  }
}
```

Screenshot



But, how?

When I started thinking about SectorC, I had just finished Deobfuscating OTCC with a lot of its ideas freshly loaded into my head. I also just had some healthy doses of justine.lol and Tom7 to inspire the absurdity of it all.

Did I think I would succeed? I suspected NO. Fit an entire C compiler in 510 bytes of instruction memory? Good luck (sarcasm).

Tokenizing

The first problem came quickly. In C, the tokenizer/lexer alone seems larger than one 512 byte sector! We need to consume an arbitrary stream of bytes and produce “tokens”.

For example:

```
int main()
{
  if( a < 5 ){
    func();
  }
}
```

Would be consumed and converted into:

```
'int'  TOKEN_KEYWORD_INT
'main' TOKEN_IDENTIFIER
'('    TOKEN_LPAREN
')'    TOKEN_RPAREN
```

```
'{'    TOKEN_LBRACE
'if'   TOKEN_KEYWORD_IF
'('    TOKEN_LPAREN
'a'    TOKEN_IDENTIFIER
'<'    TOKEN_OPERATOR
'5'    TOKEN_NUMBER
')'    TOKEN_RPAREN
'{'    TOKEN_LBRACE
'func' TOKEN_IDENTIFIER
'('    TOKEN_LPAREN
')'    TOKEN_RPAREN
';'    TOKEN_SEMI
'}'    TOKEN_RBRACE
'}'    TOKEN_RBRACE
```

We need to specifically recognize `keywords`, `identifiers`, `operators`, and `numbers`. And then we need to convert numbers from string to integer with something like `atoi()`:

```
int atoi(const char *s)
{
  int n = 0;
  while (1) {
    char c = *s++;
    if (!c) break;
    n = 10 * n + (c - '0');
  }
  return n;
}
```

I wrote a fairly straight-forward and minimalist lexer and it took >150 lines of C code. A crude estimate of the same code in x86-16 would require 300-450 bytes minimum (e.g. a simple `add ax,bx` instruction encodes as 2 bytes). And this doesn't include any symbol table, recursive-descent parser, code-generator, branch-patching, etc.

No Chance.

So, naturally … I continued. Always pick the losers. The lolz are more fun that way.

Big Insight #1

Big Insight #1 came while thinking about other languages such as Forth. The tokenizer in Forth is nearly trivial. Every token is simply space-delimited. Every token is just called a `WORD` and nothing is special (slight lie). Hmm, how about a C that does that? So dreamed up a C that is technically still a C, is probably turing-complete, and will definitely make every code maintainer terrified. 😌

I will call it the *Barely C Programming Language*:

```
int done , a , b , c , p , cond ;
int(main)(){while(!done){
     a = b - c ;
     *(int*) p = b - c ;
     a = *(int*) p ;
     if(cond) a = b - 45 ;
}}
```

Here we have spacing strategically placed to create "mega-tokens"

For example: `int(main){while(!done){` is one such "mega-token".

In a sense, we actually have a language more like:

```
VAR_BEGIN done AND a AND b AND c AND p AND cond END
MAIN_BEGIN
  a = b - c END
  DEREF p = b - c END
  a = DEREF p END
  COND a = b - 45 END
MAIN_END
```

But, a normal C compiler will also recognize it as C!

Even after using space-delimiters, we still have a lot of tokens and need to find more ways to minimize the tokenizer. What is essential? Well it's quite hard to avoid the `atoi()` if we want to actually have integer literals. What else do we need? How about nothing.

Big Insight #2

Big Insight #2 is that `atoi()` behaves as a (bad) hash function on ordinary text. It consumes characters and updates a 16-bit integer. Hashes are perhaps the holy-grail of computer-science. With a good hash, we can just side-step all the hard problems by trading them for an even harder problem (hash collisions), and then we just ignore that harder problem. Brilliant. (sticks fingers in ears) 🤪

So we have this:

Token Type	Meaning of atoi()
Integer Literal	uint16 number
Keyword	token "enum" value
Identifier	hash value into a 64K array

Implementing Barely C

The first implementation of Barely C fit in 468 bytes. It was a simple recursive-descent parser over the `atoi` tokens. There was no symbol table of any kind. Variables simply access a 64K segment using the hash value. Codegen is emitted somewhat similar to OTCC, using `ax` as the result register and shuffling values to the stack and then to `cx` for binary operators.

Minimizing with Byte-Threaded Code

In an attempt to steal every good idea Forth ever had, I then dreamed up what I will call "byte-threaded-code". Since a sector is 512 bytes, if we simply align address on a 2-byte boundary, we can do addressing with a single byte! We can have a series of "gadgets" and do forth-style threading:

```
bits 16
  cpu 386

  jmp 0x07c0:entry

entry:
  push cs
```

```
  pop  ds

  lea si,operations
next:
  xor ax,ax
  lodsb
  add ax,ax
  push next
  jmp ax

putch:
  mov ah,0x01
  mov al,bl
  mov dx,0
  int 0x14
  ret

  align 2
hang:
  jmp hang

  align 2
print_F:
  mov bx,'F'
  jmp putch

  align 2
print_G:
  mov bx,'G'
  jmp putch

operations:
  db 17  ; print_F
  db 20  ; print_G
  db 17  ; print_F
  db 17  ; print_F
  db 20  ; print_G
  db 17  ; print_F
  db 17  ; print_F
  db 17  ; print_F
  db 20  ; print_G
  db 16  ; hang
```

Annoyingly, `nasm` won't let you do something like `db print_F/2` so I had to write a custom little assembler to do it.

Alas, this idea didn't work out. In 512 bytes, the overhead of this Forth-style computation model doesn't pay for itself. There are a lot of little overheads: 2 byte alignment, extra `ret` instructions, calling other "threads", the `next` function, etc. The byte-threaded version of Barely C ended up at the same size as the straight-forward version

However, the idea is fun and I decided to document it anyways in the event that someone else finds utility.

Minimizing the Straight-Forward version

Instead, I returned to the straight-forward version and minimized it as much as possible. From 468 bytes ⇒ 303 bytes (165 bytes saving): 510 - 303 ⇒ 207 spare bytes to use for new features!

Some tricks:

- Reorganize code to allow “fall-through” instead of `jmp` or `call`
- Use tail-calls via `jmp` wherever possible
- Perform call-fusion (e.g. `call tok_next2` instead of `call tok_next; call tok_next`)
- Utilize `stosw` and `lodsw` extensively
- Eliminate machine code tables for cheaper inline `stosw` versions
- Prefer `cmp ax,imm` over `cmp bx,imm`
- Keep jump offsets within 8-bits to encode more efficiently

Look Ma, A Real C!

As it turns out, a lot can be accomplished in 200 bytes if you already have a tokenizer, parser, and code-generator in 300 bytes. With these 200 bytes, *Barely C* became a proper C:

- Arbitrarily nested `if` statement block with an arbitrary expression condition
- Arbitrarily nested `while` statement block with an arbitrary expression condition
- Lots of operators: `+`, `-`, `*`, `&`, `|`, `^`, `<<`, `>>`, `==`, `!=`, `<`, `>`, `<=`, `>=`
- Grouping expressions: `( expression )`
- Function definitions and recursive function calls (using `func()` as a hash value into a symbol table at segment 0x3000)
- A special `asm` statement for inline machine-code
- Single-line `//` comments
- Multi-line `/*` comments
- A trick to do “space-injection” before semicolons to make code look more normal

The biggest enabler here is the `binary_oper_tbl` which allows for a very cheap way to add lots of operations. Each operator is simply a `<16-bit token-value> <16-bit-machine-code>` pair, costing just 4 bytes.The above 14 operators cost just 56 bytes plus a little overhead to scan the table.

Grammar

Here's the full grammer specification:

```
program     = (var_decl | func_decl)+
var_decl    = "int" identifier ";"
func_decl   = "void" func_name "{" statement* "}"
func_name   = <identifier that ends in "()" with no space>
statement   = "if(" expr "){" statement* "}"
            | "while(" expr "){" statement* "}"
            | "asm" integer ";"
            | func_name ";"
            | assign_expr ";"
assign_expr = deref? identifier "=" expr
deref       = "*(int*)"
expr        = unary (op unary)?
unary       = deref identifier
            | "&" identifier
            | "(" expr ")"
            | indentifier
            | integer
op          = "+" | "-" | "&" | "|" | "^" | "<<" | ">>"
            | "==" | "!=" | "<" | ">" | "<=" | ">="
```

In addition, both `// comment` and `/* multi-line comment */` styles are supported.

(NOTE: This grammar is 704 bytes in ascii, 38% larger than it's implementation!)

Inline Machine-Code

A programming language without I/O is useless. And, as the C language is defined in an I/O agnostic way, we need some way out. Thus, an `asm` extension is supported. This allows programs to generate raw x86-16 machine code literals inline. Using `asm`, programs can access any low-level detail of the machine. This is used extensively in the example code.

Error-Handling

What is “error-handling”? 🤣

In traditional C style, we trust the programmer to write correct and well-formed programs. We are certain they are all minor gods and goddesses with the ability of perfection. Obviously, spending bytes on error-checking would be foolish. Surely all will agree that this is a very reasonable standard.

For the less divine among us, a `lint` was also written (that doesn’t fit in a sector) to detect errors. The author certainly didn’t require this tool for development.

Runtime

If C compiler writers were a secret shadow organization like the Free Masons, Illuminati, Lizard Peoples, or Pizzagaters our inner-secret would be “C actually has a runtime”.

SectorC has a runtime under `rt/` consisting of two files implemented in C itself:

- `rt/lib.c`: A collection of library routines, often coded in inline `asm`
- `rt/_start.c`: The actual entry-point `_start()`

The runtime code is concatenated with program source to construct the full source to compile and run.

Examples

A few examples are provided that leverage the unique hardware aspects of the x86-16 IBM PC:

- `examples/hello.c`: Print a text greeting on the screen writing to memory at `0xB8000`
- `examples/sinwave.c`: Draw a moving sine wave animation with VGA Mode 0x13 using an appropriately bad approximation of `sin(x)`
- `examples/twinkle.c`: Play “Twinkle Twinkle Little Star” through the PC Speaker (Warning: LOUD)

Conclusion

It seems fitting to end an article with “takeaways” or “what did we learn”. So.. umm.. what did we learn? Honestly, I’m not sure. But in the interest of fun, here’s a Choice Your Own Adventure version of What Did We Learn:

What Did We Learn	Your Chosen Adventure
Things that seem impossible often aren’t and we should Just Do It anyway	Move to the South Pole with absolutely no gear on a Homesteading Mission
Software is too bloated these days, we only need a few KBs	Go check yourself into the technology hippie-commune of suckless

What Did We Learn	Your Chosen Adventure
Error checking is overrated	Take Elon up on his pitch to be a mars astronaut because Earth really doesn't need more software that ignores errors.
Anything X can do, C can do better	Something like this? link. Monzy, we need a new rap! (call me)
That was all gibberish nonsense and thank you for wasting my life (passive-aggression)	Feel regret that you wasted the time because there is a lot better content in the world you'd rather consume and decide to get a therapist to work through your issues with reading nonsense internet gibberish
This xorvoid person/robot/AI is ridiculous/absurd/dumb and does arguably pointless things for fun	Follow, like, subscribe, ring the bell.. 😁 (rss)
