

C – Why use it?

Paul Schrimpf

January 16, 2009

Why Use C?

- It is harder to learn and use
- Fast
- Ubiquitous
 - ▶ C can be combined with almost any other language
 - ▶ Many algorithms have been written in C that are not available in Matlab and Stata

Example: Consumption and Bequests

- Estimation for Einav, Finkelstein, and Schrimpf (2007) required solving the following problem tens of thousands of times:

$$V(w_0, g, \alpha, \beta) = \max_{c_t, w_t} \sum_{t=0}^T s_t(\alpha) \delta^t \frac{c_t^{1-\gamma}}{1-\gamma} + \beta m_t(\alpha) \delta^t \frac{(w_t + G_t^g)^{1-\gamma}}{1-\gamma}$$
$$\text{s.t. } 0 \leq w_{t+1} = (w_t + z_t^g - c_t)(1+r)$$

- Strategy: use first order conditions to concentrate out $c_t(c_0)$, $w_t(c_0)$ and then use Brent's method to maximize over c_0

$$\lambda_0 = s_0(\alpha) \delta^0 c_0^{-\gamma}$$

$$w_{t+1} = (w_t + z_t^g - c_t)(1+r)$$

$$\lambda_{t+1} = -m_{t+1}(\alpha) \delta^{t+1} \beta (w_{t+1} + G_{t+1}^g)^{-\gamma} + \frac{1}{1+r} \lambda_t$$

$$c_t = \left(\frac{\lambda_t}{\delta^t s_t(\alpha)} \right)^{-1/\gamma}$$

Example (continued)

- Natural to program as a loop \rightarrow very slow in Matlab compared to C
- Solving 100 times in Matlab with objective function written in Matlab takes 6 seconds
- With the objective function written in C it takes 0.1 seconds
- C is sixty times faster even though algorithm is identical and the code looks very similar

Demonstration

Why is C so much faster?

- Consider

```
1  for i=1:100
2      x(i) = i;
3  end
```

- Each iteration Matlab will:

- 1 Increment i , check if $i \leq 100$
- 2 Check that i is a valid index for x
- 3 Check that $x(i)$ is allocated, allocate it if necessary
- 4 Check that $x(i)$ is a valid target for i
- 5 Copy the value of i into $x(i)$, change type of i if needed

Why is C so much faster?

- Equivalent C code:

```
1  for ( i=0; i < 100; i++) {  
2      x[i] = i;  
3  }
```

- Each iteration C will:
 - ① Increment i , check if $i \leq 100$
 - ② Copy the value of i into $x(i)$, change type of i if needed
- C only does what you tell it to \rightarrow fast, but also difficult and error prone

Major Practical Difference Between C and Matlab

- Must declare variables
 - ▶ Names and sizes – must deal with dynamic memory management yourself
- C has few functions builtin – need to link to other libraries
- C is compiled, Matlab is interpreted (or JIT compiled)
- C does exactly what you tell it, even if you tell it to do something bad

References

- You will need them
- [The C Book](#) – great introductory reference
 - ▶ “Programming in C is like eating red meat and drinking strong rum except your arteries and liver are more likely to survive it.”
- Kernighan and Ritchie (K&R) – classic but not great for learning
 - ▶ [Kernighan: Programming in C: A Tutorial \(1974\)](#) – historical
- [C Programming Notes](#) – K&R explained
- [Common C Errors](#)

Quotes

- “C is often described, with a mixture of fondness and disdain varying according to the speaker, as ‘a language that combines all the elegance and power of assembly language with all the readability and maintainability of assembly language’ ” – Jargon File (MIT and Stanford AI labs circa 1983)
- “C is quirky, flawed, and an enormous success.” – Dennis Ritchie
- “C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg.” – Bjarne Stroustrup

Good News

- You already know some C
- Matlab's structs, `printf`, `strcmp`, etc. take their names and functionality from C
- Perl's flow control, indexing scheme, etc. is also very similar to C

C Datatypes

- Declaring datatypes
- Every variable must be declared before it can be used

```
1 float y;    // single precision scalar, 8 digits, rarely u
2 double x;   /* double precision scalar, 16 digits, 64 bits
3 long double w; /* more precise double, 96 bits */
4 int i;      /* integer, b/t  $-2^{(31)}$  and  $2^{(31)}$  */
5 short int j; /* 16 bit integer */
6 char a,b,c;  /* 8 bit characters – each holds 1 letter
7 unsigned int u; /* integer b/t 0 and  $2^{32}$  */
```

- Sizes and ranges of datatypes are machine dependent
- Can only declare variables in certain places – start of of a { } block

Basic Operators

```
1  double x,y=1,z=2;
2  /* usual +,*,/, - */
3  x = y+z; y = z*x;  z = y/x;
4
5  /* less familiar: */
6  x = 1;
7  x++; /* now x is 2; */
8  y = x--; /* now y=2, x=1 */
9  z = ++x; /* now z=2, x=2 */
10 y /= 4; /* sets y = y/4; */
11 x = y>z? 1:2; /* x = 2, because y>z is false */
```

Mixing Datatypes

```
1  double x;  
2  int n, j=3, k=6;  
3  char c;  
4  
5  n = 2*x; /* ???, but not an error */  
6  n = k%j; /* modulo */  
7  x = j/k; /* x=0 b/c j/k is an int */  
8  x = ((double) j)/((double) k); /* now, x=0.5 */  
9  n = x; /* n = 0 b/c it cast x into an int */  
10  
11 c = x*j; /* ???, but not an error */  
12 x = (j=(6+k=7)); /* valid, but confusing */
```

Logical Operators

- `==` , `!=` , `>=` , `>` , `<=` , `<` , `!` , `||` , `&&`
- Any non-zero value is true as part of a logical expression
- In arithmetic, true logical expressions act like `int true=1`
- Bitwise – do not use on accident
`&` , `|` , `^` , `>>` , `<<` , `~`

Flow Control

- Same type of commands as Matlab:

```
1  for(initialization; stopping condition; increment) {  
2      /* braces are optional if we just have single command */  
3      doSomething;  
4  }
```

```
1  do {  
2      doSomething at least once;  
3  } while (condition); /* semicolon required here /
```

```
1  while (condition){  
2      doSomething;  
3  }
```

Flow Control

```
1  if (condition) {  
2      doSomething;  
3  } else { /* can leave out else, add else if, etc */  
4      doSomethingElse  
5  }
```


More Flow Control

- To be used sparingly:

```
1  switch (integer) {
2  case constant1:
3      command1;
4  case constant2:
5      command2;
6      break; /* note: case constant1 executes
7              both command1 and command2 */
8  default:
9      command3;
10 }
```

```
1  while(condition) {
2      command1;
3      if (cond2) break; /* exit loop early */
4      else if (cond3) continue; /* go to start of loop */
5      command2;
6  }
```

Why we fear break – AT&T 1990 Outage

- Switches had this type of code:

```
1  do {
2      /* ... many lines omitted ... */
3      switch (something) {
4          case 1:
5              /* ... many lines omitted ... */
6              if (wantToStop) break;
7              /* ... many lines omitted ... */
8              break;
9          case 2:
10             /* ... many lines omitted ... */
11         }
12     } while (condition);
```

- This code caused big problems ...

Why we fear break – AT&T 1990 Outage

- When switches crash they send an “out of service” message
- This code made it so that when a switch received an “out of service” message, it crashed ... and then rebooted
- Result: one switch failed, nearby switches crashed, making more switches crash, ... switches reboot in time to receive to the “out of service” message from far away switches, so they immediately crash again
 - ▶ Took 9 hours to fix
 - ▶ Cost \$60 million

More Flow Control

- To be used sparingly:

```
1  if (condition) goto label1; /* the infamous goto */
2  /* ... bunch of stuff to skip ... */
3  label1: /* got sent here
```

- “Everybody knows that the goto statement is a ‘bad thing.’ Used without care it is a great way of making programs hard to follow and of obscuring any structure in their flow. Dijkstra wrote a famous paper in 1968 called ‘Goto Statement Considered Harmful,’ which everybody refers to and almost nobody has read.” — [The C Book](#)

References

This section go over the simplest way to compile a C program. For more information see:

- [ACM Developing Software in Unix Tutorial](#) – everything you need to get started in Unix or Linux
- [GCC manual](#)

Getting a Compiler

- Many compilers are available – gcc is the most common, Intel's icc can be faster, many others exist
- To install
 - ▶ Linux / Unix – already have it
 - ▶ Windows
 - ★ IDE – Bloodshed Dev-C++
 - ★ Unix-like – MINGW, Cygwin
 - ▶ Mac OS X – Xcode
- If you just want to use within Matlab, try typing mex, it may work
- If you don't use an IDE, you'll want a quality editor too – Emacs and Vi(m) are classics
- My preference: Linux or MINGW & Emacs

Hello, World!

```
1 #include <stdio.h>
2
3 int main () {
4     printf("Hello , World!\n");
5 }
```

- ❶ Open your editor, type it, and save it as helloWorld.c
 - ❷ To compile: `gcc helloWorld.c -o hello.exe`
 - ❸ To run: `./hello.exe`
- Fun fact: A Tutorial Introduction to the Language B (Kernighan, 1972) – contains the original “Hello World!”

Hello, Matlab!


```
1 #include "mex.h"
2
3 void mexFunction(int nlhs, mxArray *plhs[],
4                  int nrhs, const mxArray *prhs[])
5 {
6     mexPrintf("Hello , Matlab!\n");
7 }
```

- 1 To compile: `mex helloWorld.c` (within Matlab)
- 2 To run: `helloWorld()` (within Matlab)

Hello, Stata!

```
1 #include "stplugin.h"
2
3 STDCALL stata_call(int argc, char *argv[])
4 {
5     SF_display("Hello Stata!\n");
6     return(0);
7 }
```

- 1 Download stplugin.c and stplugin.h¹
- 2 To compile: `gcc -ansi -shared -fPIC [-DSYSTEM=OPUNIX] stplugin.c helloStata.c -o hello.plugin`
- 3 To use, in stata:
program hello, plugin
plugin call hello

¹See <http://www.stata.com/plugins/> for more information. 

Hello What?

```
1  int i; main(){ for (; i[' ' < i; ++i){ --i; } ' ' ]; read( '- - - ', i+++ ' ' hell \
2  o, world! \n ' ', ' / ' / ' / ' )); } read(j, i, p){ write(j/p+p, i — j, i/i); }
```

- Winner of the 1984 International Obfuscated C Code Contest

Example: Fibonacci Sequence

- We'll demonstrate the tools we've covered so far by writing a function that computes the n th Fibonacci number

$$F(n) = \begin{cases} n & \text{if } n = 0, 1 \\ F(n-2) + F(n-1) & \text{if } n > 1 \end{cases} \quad (1)$$

- We'll compute $F(n)$ naïvely – the clever method is left as an exercise

Fibonacci: Loop Version

```
1 unsigned int fibLoop(unsigned int n) {
2     unsigned int F,Fm1,Fm2;
3     int i;
4
5     if (n≤1) return(n);
6     Fm1=1;
7     Fm2=0;
8     for(i=2; i≤n; i++) {
9         F = Fm2+Fm1;
10        Fm2 = Fm1;
11        Fm1 = F;
12    }
13    return(F);
14 }
```

Fibonacci: Recursive Version

```
1 unsigned int fibRecur(unsigned int n) {  
2     if (n≤1) return(n);  
3     else return(fibRecur(n-1) + fibRecur(n-2));  
4 }
```

- More elegant
- Much more time consuming for large n – big call stack

To Use from Command Line

```
1 #include <stdio.h> /* header file for input/output functions
2 /* function declarations — needed so we can use them in main
3 unsigned int fibLoop(unsigned int);
4 unsigned int fibRecur(unsigned int);
5
6 int main(int argc, char **argv) {
7     unsigned int n;
8     if (argc != 2) {
9         printf("ERROR: 1 command line argument required\n");
10        return(12); /* should return 0 on success, non-zero on f
11    }
12    if (!sscanf(argv[1], "%d", &n)) {
13        printf("ERROR: bad input \"%s\"\\n", argv[1]);
14        return(13); /* should return 0 on success, non-zero on fa
15    }
16    printf("Loop version: The %d Fibonacci number is %d\\n", n, fi
17    printf("Recursive version: The %d Fibonacci number is %d\\n"
18    return(0);
19 }
```

To Use in Matlab

```
1 #include "mex.h" /* header file for matlab API */
2
3 unsigned int fibLoop(unsigned int);
4
5 void mexFunction
6 (int nlhs, /* number of left hand side arguments */
7  mxArray *plhs[], /* pointer to lhs*/
8  int nrhs, /* number of rhs arguments*/
9  const mxArray *prhs[]) /* pointer to rhs arguments*/
10 {
11     /* error checking omitted */
12
13     plhs[0] = mxCreateDoubleScalar(fibLoop(mxGetScalar(prhs[0])))
14 }
```

To Use in Stata

```
1  STDCALL stata_call(int argc, char *argv[])
2  {
3      ST_int      j;
4      ST_double   val, fib;
5      ST_retcode  rc ;
6      for(j = SF_in1(); j ≤ SF_in2(); j++) {
7          if(SF_ifobs(j)) { /* true if stata if command true */
8              if(rc = SF_vdata(1,j,&val)) return(rc) ;
9              fib = (double) fibLoop((unsigned int) val);
10             if(rc = SF_vstore(SF_nvars(), j, fib)) return(rc) ;
11         }
12     }
13     return(0) ;
14 }
```


Lots More to Learn ...

- Preprocessor directives, e.g. `#include`
- Header files, e.g. `"mex.h"`
- More datatypes, e.g. `const mxArray`
- Pointers, e.g. `char **argv`

The C Preprocessor

- Before compiling, a C source code is preprocessed, *i.e.* the text is modified
- Preprocessor commands begin with `#`
- `#define` `PI 3.14` defines a constant
- `#define` `MAX(a,b) a ≥ b ? a : b` defines a macro
- `#ifdef` `PI` *some command* `#endif` only includes *some command* if the macro `PI` had been defined
- `#include` `<file>` or `#include` `"file"` inserts the contents of *file*

Header Files

- Contain function and variable declarations – must declare functions before you can use them
- e.g. `<stdio.h>` declares `scanf()` and `printf()`
`"mex.h"` declares `mexArray`, `mexGetPr()`, etc
- Files inside `<>` are supposed to be system header files that are part of your compiler or OS
 - ▶ Common headers: `<math.h>`, `<stdio.h>`, `<stdlib.h>`, etc.
 - ▶ Use `man header.h` to learn about header file or `man function` to learn about a function
- Only contain function prototypes, not the actual code
- Actual functions are contained in a library file somewhere

Compilation

- 1 Preprocessing
- 2 Compiler – changes source files to assembly language

```
1    mov  eax , [L1]      ; eax = byte at L1
2    mov  ebx , [L2]      ; ebx = byte at L2
3    add  eax , ebx       ; eax = eax + ebx
4    mov  [L1] , eax      ; byte at L1 = eax
```

- 3 Assembler – translates assembly into machine language
 - 4 Linker – examines object file(s), creates list of functions/variables referenced but not defined, and either:
 - ▶ static linking: searches system libraries for them, adds the ones found, and finally packages everything into an executable
 - ▶ dynamic linking: inserts commands that load the needed libraries when the program is executed
- Usually do not have to worry about details, just type:
 - ▶ Command line: `gcc [-g] -Wall file1.c file2.c -o program.exe`
 - ▶ Matlab: `mex [-g] file.x`

Compiler Options

- Compiling can get very complicated
 - ▶ Hundreds of options – `man gcc` is 6884 lines
 - ▶ Linking libraries
 - ▶ Breaking into steps
- Tools to manage
 - ▶ IDE – automates much, menus for options
 - ▶ `make` – manages complicated compilations [A Make Tutorial](#)
- Important options:
 - ▶ `-g` turns on debugging information
 - ▶ `-Wall` turns on all warning messages
 - ▶ `-l/lib` tells to link against library `lib` – e.g. for math, `-lm`
 - ▶ `-O2` and `-O3` adds optimizations – can do even better with fine tuning

Arrays

```
1  double x[10], y[5][6];  
2  int i;  
3  for(i=0; i<10; i++) x[i] = sqrt(i);
```

- Indexed starting with 0
 - ▶ “Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration.” – Stan Kelly-Bootle
- **No** way to figure out size

- structs and typedef

```
1  struct vector_s {
2      unsigned int size; /* size */
3      double *val; /* pointer to array of values */
4  };
5  typedef struct vector_s vector; /* define a new variable
6                                   "vector" */
7  vector x; /* declare x to be a vector */
8  /* use like in Matlab */
9  x.size = 10; /* etc */
10 x.val = malloc(x.size*sizeof(double));
```

Enums

- enums

```
1  typedef enum {                /* a named multinomial variable */
2      beer=1, wine, milk        /* make beer=1, wine=2, milk=3 */
3  } beveridge;                  /* name this enum beveridge */
4  beveridge cup;
5  switch(cup) {
6      case beer: chug(cup);      break;
7      case wine: sip(cup);       break;
8      case milk: drink(cup);     break;
9      default:   poison(cup);
10 }
```


Unions

- unions – exotic

```
1  union {           /* declare like structs      */
2      double x;      /* but, the fields share the */
3      int i;         /* same place in memory     */
4  } u;
5  u.x = 3.5;         /* if we examine u.i it would be something s
6  u.i = 2;           /* u.x has now changed to something strange
```

Variable Scope and Type Modifiers

- Normally local within {}
- Outside of any function → accessible by all functions in that file
- Prefixes change behaviour
 - ▶ `const int` `x = 3` makes `x` constant
 - ▶ `static double` `x` makes `x` static / persistent
 - ▶ `extern double` `x` makes `x` global
- Prefixes can have different meaning depending on context
 - ▶ `double` `function(const double x)` means that `x` cannot be modified within function
 - ▶ `static void` `f(char *s)` means that `f()` can only be called by other functions in the same file
- Exotic type modifiers – `volatile` and `register`

Exercises

Unlike the Matlab portion of the course, it will be very difficult to get much out of this without trying the exercises. At the very least, you should attempt to compile and run the provided code. I've wasted many days struggling with compiler related configuration problems.

- 1 A good algorithm can calculate the Fibonacci sequence in $O(\log_2 n)$ steps. Develop and implement one.
- 2 Try giving passing some strange arguments to the Fibonacci programs, e.g. non-integers, negative numbers, imaginary numbers, characters, etc. What happens? (It's quite likely that the program will enter an infinite loop. Press `^C` to stop the command line version. You may have to kill Matlab. Use Windows' task manager, or type "killall -TERM MATLAB" at linux command line.) Modify the program so that it doesn't behave so badly.