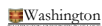


Introduction to the C Programming Language

Fred Kuhns
fredk@cse.wustl.edu
Applied Research Laboratory,
Department of Computer Science and Engineering,
Washington University in St. Louis



Introduction

- The C programming language was designed by Dennis Ritchie at Bell Laboratories in the early 1970s
- Influenced by
 - ALGOL 60 (1960),
 - CPL (Cambridge, 1963),
 - BCL (Martin Richard, 1967),
 - B (Ken Thompson, 1970)
- Traditionally used for systems programming, though this may be changing in favor of C++
- Traditional C:
 - *The C Programming Language*, by Brian Kernighan and Dennis Ritchie, 2nd Edition, Prentice Hall
 - Referred to as *K&R*

Standard C

- Standardized in 1989 by ANSI (American National Standards Institute) known as ANSI C
- International standard (ISO) in 1990 which was adopted by ANSI and is known as *C89*
- As part of the normal evolution process the standard was updated in 1995 (*C95*) and 1999 (*C99*)
- C++ and C
 - C++ extends C to include support for Object Oriented Programming and other features that facilitate large software development projects
 - C is not strictly a subset of C++, but it is possible to write "Clean C" that conforms to both the C++ and C standards.

Elements of a C Program

- A C development environment includes
 - *System libraries and headers*: a set of standard libraries and their header files. For example see /usr/include and glibc.
 - *Application Source*: application source and header files
 - *Compiler*: converts source to object code for a specific platform
 - *Linker*: resolves external references and produces the executable module
- User program structure
 - there must be one main function where execution begins when the program is run. This function is called main
 - int main (void) { ... },
 - int main (int argc, char *argv[]) { ... }
 - UNIX Systems have a 3rd way to define main(), though it is not POSIX.1 compliant
 - int main (int argc, char *argv[], char *envp[])
 - additional local and external functions and variables

Fred Kuhns (1/26/2004)

CS422 - Operating Systems Concepts

4

A Simple C Program

- Create example file: try.c
- Compile using gcc:
gcc -o try try.c
- The standard C library *libc* is included automatically
- Execute program
./try
- Note, I always specify absolute path
- Normal termination:
void **exit**(int status);
 - calls fncs registered with atexit()
 - flush output streams
 - close all open streams
 - return status value and control to host environment

```
/* you generally want to include
   stdio.h and stdlib.h */
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    printf("HelloWorld\n");
    exit(0);
}
```

Fred Kuhns (1/26/2004)

CS422 - Operating Systems Concepts

5

Source and Header files

- Just as in C++, place related code within the same module (i.e. file).
- *Header files*: exported interface definitions (function prototypes, data types, macros, inline functions and other common declarations)
- *Do not place source code in the header file*: but you can place inline'd code in the header file.
- *C preprocessor (cpp)*: used to insert common definitions into source files
- There are other cool things you can do with the preprocessor

Fred Kuhns (1/26/2004)

CS422 - Operating Systems Concepts

6

The Preprocessor

- The C preprocessor permits you to define simple macros that are evaluated and expanded prior to compilation.
- Commands begin with a '#'. Abbreviated list:
 - #define : defines a macro
 - #undef : removes a macro definition
 - #include : insert text from file
 - #if : conditional based on value of expression
 - #ifdef : conditional based on whether macro defined
 - #ifndef : conditional based on whether macro is not defined
 - #else : alternative
 - #elif : conditional alternative
 - defined : preprocessor function: 1 if name defined, else 0

Preprocessor: Macros

- Using macros as functions, exercise caution:
 - flawed example: #define mymult(a, b) a * b
 - Source: k = mymult(i-1, j+5);
 - Post preprocessing: k = i - 1 * j + 5;
 - better: #define mymult(a, b) (a) * (b)
 - Source: k = mymult(i-1, j+5);
 - Post preprocessing: k = (i - 1) * (j + 5);
- Be careful of *side effects*, for example what if we did the following
 - Macro: #define mysq(a) (a) * (a)
 - flawed usage
 - Source: k = mysq(++)
 - Post preprocessing: k = (++)* (++)
- Alternative is to use inline'd functions
 - inline int mysq(int a) {return a*a};
 - mysq(++)

Preprocessor: Conditional Compilation

- Its generally better to use inline'd functions
- Typically you will use the preprocessor to define constants, perform conditional code inclusion, include header files or to create shortcuts
- #define DEFAULT_SAMPLES 100
- #ifdef __linux
 - static inline int64_t gettime(void) {...}
- #elif defined(sun)
 - static inline int64_t gettime(void) {return (int64_t)gethrtime()};
- #else
 - static inline int64_t gettime(void) {... gettimeofday()...}
- #endif

Pointers Continued

Step 1:

```
int main (int argc, argv) {
    int X = 4;
    int *Y = &X;
    int *Z[4] = {NULL, NULL, NULL,
    NULL};
    int a[4] = {1, 2, 3, 4};
}
```

Step 2:

```
Z[0] = a;
Z[1] = a + 1;
Z[2] = a + 2;
Z[3] = a + 3;
```

Step 3: No change in Z's values

```
Z[0] = (int *)((char *)a);
Z[1] = (int *)((char *)a + sizeof(int));
Z[2] = (int *)((char *)a + 2 * sizeof(int));
Z[3] = (int *)((char *)a + 3 * sizeof(int));
```

	Program Memory	Address
X	4	0x3dc
Y	0x3dc	0x3d8
	NA	0x3d4
	NA	0x3d0
Z[3]	0x3bc	0x3cc
Z[2]	0x3b8	0x3c8
Z[1]	0x3b4	0x3c4
Z[0]	0x3b0	0x3c0
a[3]	4	0x3bc
a[2]	3	0x3b8
a[1]	2	0x3b4
a[0]	1	0x3b0

Pointers

- For any type T, you may form a pointer type to T.
 - Pointers may reference a function or an object.
 - The value of a pointer is the address of the corresponding object or function
 - Examples: `int *i; char *x; int (*myfunc)();`
- Pointer operators
 - * dereferences a pointer, & creates a pointer (reference to)
 - `int i = 3; int *j = &i;`
`*j = 4; printf("i = %d\n", i); // prints i = 4`
 - `int myfunc (int arg);`
`int (*fptr)(int) = myfunc;`
`i = fptr(4); // same as calling myfunc(4);`
- Generic pointers:
 - Traditional C used (`char *`)
 - Standard C uses (`void *`) - these can not be dereferenced or used in pointer arithmetic. So they help to reduce programming errors
- Null pointers: use NULL or 0. It is a good idea to always initialize pointers to NULL.

Arrays and Pointers

- All arrays begin with an index of 0
- An array identifier is equivalent to a pointer that references the first element of the array
 - `int a[4], *ptr;`
`ptr = &a[0]` is equivalent to `ptr = a;`
- Pointer arithmetic and arrays:
 - `int a[10];`
`a[5]` is the same as `(a + 5)`, the compiler will assume you mean 5 objects beyond element `a`.

Functions

- Always use function prototypes
 - int myfunc (char *, int, struct MyStruct *);
 - int myfunc_noargs (void);
 - void myfunc_noreturn (int i);
- Call by value, copy of parameter passed to function
 - if you want to alter the parameter then pass a pointer to it
 - Note: arrays will be passed by reference
- If performance is an issue then use inline functions, generally better and safer than using a macro. Common convention
 - define prototype and function in header or *name.i* file
 - static inline int myinfunc (int i, int j);
 - static inline int myinfunc (int i, int j) { ... }

Basic Types and Operators

- Basic data types
 - Types: char, int, float and double
 - Qualifiers: short, long, unsigned, signed, const
- Constant: 0x1234, 12, "Some string"
- Enumeration:
 - Names in different enumerations must be distinct
 - enum WeekDay_t {Mon, Tue, Wed, Thur, Fri};
 - enum WeekendDay_t {Sat = 0, Sun = 4};
- Arithmetic: +, -, *, /, %
 - prefix ++ or --: increment/decrement before value is used
 - postfix i++, i--: increment/decrement after value is used
- Relational and logical: <, >, <=, >=, ==, !=, &&, ||
- Bitwise: &, |, ^ (xor), <<, >>, ~(ones complement)

Operator Precedence

Operators	Category	Associativity
() [] -> . ++ --	Postfix expressions	left to right
! ~ ++ -- + - * & (type) sizeof	Unary operators	right to left
* / %	Multiplicative operators	left to right
+ -	Additive operators	left to right
<< >>	Shift operators	left to right
<<= >>=	Relational operators	left to right
== !=	Equality operators	left to right
&	Bitwise AND operator	left to right
^	Bitwise XOR operator	left to right
	Bitwise OR operator	left to right
&&	Logical AND operator	left to right
	Logical OR operator	left to right
?:	Conditional (<i>ternary</i>) operator	right to left
= += -= *= /= %= &= ^= = <<= >>=	Assignment expressions	right to left
,	Comma operator	left to right

Structs and Unions

- **structures**
 - `struct MyPoint {int x, int y};`
 - `typedef struct MyPoint MyPoint_t;`
 - `MyPoint_t point, *ptr;`
 - `point.x = 0; point.y = 10;`
 - `ptr = &point; ptr->x = 12; ptr->y = 40;`
- **unions**
 - `union MyUnion {int x; MyPoint_t pt; struct {int 3; char c[4]};`
 - `union MyUnion x;`
 - Can only use one of the elements. Memory will be allocated for the largest element

Conditional Statements (if/else)

- `if (a < 10)`
`printf("a is less than 10\n");`
`else if (a == 10)`
`printf("a is 10\n");`
`else`
`printf("a is greater than 10\n");`
- If you have compound statements then use brackets (blocks)
- `if (a < 4 && b > 10) {`
`c = a * b;`
`printf("a = %d, a's address = 0x%08x\n", a, (uint32_t)&a);`
`b = 0;`
`} else {`
`c = a + b; b = a;`
`}`
- `if (a) x = 3; else if (b) x = 2; else x = 0;`
Is the same as
`if (a) x = 3; else {if (b) x = 2; else x = 0;}`
- **Is this correct?**
`if (a) x = 3; else if (b) x = 2; else (z) x = 0; else x = -2;`

Conditional Statements (switch)

- `int c = 10;`
`switch (c) {`
`case 0:`
`printf("c is 0\n");`
`break;`
`...`
`default:`
`printf("Unknown value of c\n");`
`break;`
`}`
- What if we leave the break statement out?
- Do we need the final break statement on the default case?

Loops

- **for** (i = 0; i < MAXVALUE; i++) {
 dowork();
}
- **while** (c != 12) {
 dowork();
}
- **do** {
 dowork();
} **while** (c < 12);
- flow control
 - break - exit innermost loop
 - continue - perform next iteration of loop
- Note, all these forms permit one statement to be executed. By enclosing in brackets we create a block of statements.

Building your program

- For all labs and programming assignments:
 - you must supply a make file
 - you must supply a README file that describes the assignment and results. This must be a text file, no MS word.
 - of course the source code and any other libraries or utility code you used
 - you may submit plots, they must be postscript or pdf

Makefiles, Overview

- Why use make?
 - convenience of only entering compile directives once
 - make is smart enough (with your help) to only compile and link modules that have changed or which depend on files that have changed
 - allows you to hide platform dependencies
 - promotes uniformity
 - simplifies my (and hopefully your) life when testing and verifying your code
- A makefile contains a set of rules for building a program

```
target ... : prerequisites ...
    command
    ...
```
- Static pattern rules, each target is matched against target-pattern to derive stem which is used to determine prereqs (see example)

```
targets ... : target-pattern : prereq-patterns ...
    command
    ...
```

Makefiles

- Defining variables
MyOPS := -DWITH
MyDIR ?= /home/fred
MyVar = \$(SHELL)
- Using variables
MyFLAGS := \$(MyOPS)
- Built-in Variables
 - \$@ = filename of target
 - \$< = name of the first prerequisites
- Patterns
 - use % character to determine stem
 - foo.o matches the pattern %.o with foo as the stem.
 - foo.o moo.o : %.o : %.c # says that foo.o depends on foo.c and moo.o depends on moo.c

Makefiles

```
MyOS := $(shell uname -s)
MyID := $(shell whoami)
MyHost := $(shell hostname)
AR := /usr/bin/ar
RANLIB := /usr/bin/ranlib
WARNSTRIC := -W -Wstrict-prototypes -Wmissing-prototypes
WARNLIGHT := -Wall
WARN := $(WARNLIGHT)
ALLFLAGS := -D_GNU_SOURCE -D_REENTRANT -D_THREAD_SAFE

APPFLGS :=
APPGFLGS = $(APPFLGS) $(ALLFLAGS) $(WARN)

WULIB := wu
WULIBDIR := $(WUSRC)/wulib
WULIBNAME := lib$(WULIB).a
WUKLIBNAME := lib$(WULIB).a
WULIBOBJDIR := $(WULIBDIR)/$(OBJDIR)
WULIBOBJ := $(WULIBOBJDIR)/$(WULIBNAME)

WUCC := gcc
WUCFLAGS := -DMyOS=$(MyOS) $(OSFLAGS) $(ALLFLAGS) $(WARN)
WUINC := -I$(WUSRC)
WUINCLUDES := $(WUINC) $(USRINC)
WULIBS := -L$(WULIBOBJDIR) -l$(WULIB) -lm
```

Example Makefile for wulib

```
include ../Makefile.inc

HDRS := log.h loopt.h net.h period.h timer.h \
stats.h bits.h keywords.h queue.h \
pqueue.h syn.h
SRCS := log.c net.c period.c pqueue.c stats.c timer.c
OBJS := $(addprefix $(OBJDIR)/, $(patsubst %.c,%.o,$(SRCS)))

all: $(OBJDIR)/$(OBJDIR)/$(WULIBNAME)

install: all

$(OBJDIR):
mkdir $(OBJDIR)

$(OBJS): $(OBJDIR)/%.o : %.c $(HDRS)
$(WUCC) $(WUCFLAGS) $(WUINCLUDES) -o $@ -c $<

$(OBJDIR)/$(WULIBNAME) : $(OBJS)
echo $(OBJDIR)
$(AR) rc $@ $(OBJS)
$(RANLIB) $@

clean:
/bin/rm -f $(OBJS)
/bin/rm -f $(KOBJS)
```

Project Documentation

- README file structure
 - **Section A: Introduction**
describe the project, paraphrase the requirements and state your understanding of the assignments value.
 - **Section B: Design and Implementation**
List all files turned in with a brief description for each. Explain your design and provide simple psuedo-code for your project. Provide a simple flow chart of you code and note any constraints, invariants, assumptions or sources for reused code or ideas.
 - **Section C: Results**
For each project you will be given a list of questions to answer, this is where you do it. If you are not satisfied with your results explain why here.
 - **Section D: Conclusions**
What did you learn, or not learn during this assignment. What would you do differently or what did you do well.

Attacking a Project

- **Requirements and scope:** Identify specific requirements and or goals. Also note any design and/or implementation environment requirements.
 - knowing when you are done, or not done
 - estimating effort or areas which require more research
 - programming language, platform and other development environment issues
- **Approach:** How do you plan to solve the problem identified in the first step. Develop a prototype design and document. Next figure out how you will verify that you did satisfy the requirements/goals. Designing the tests will help you to better understand the problem domain and your proposed solution
- **Iterative development:** It is good practice to build your project in small pieces. Testing and learning as you go.
- **Final Touches:** Put it all together and run the tests identified in the approach phase. Verify you met requirements. Polish you code and documentation.
- **Turn it in.**
