SIPB's IAP Caffeinated Crash Course in C

The classic

```
#include <stdio.h>

main()
{
  printf("Hello, world!\n");
}
```

A typical interaction with gcc, the GNU C compiler and a timeless classic in the history of C compilation.

```
athena% gcc hello-world.c -o hello-world
hello-world.c: In function ‘main’:
hello-world.c:6: error: expected ’;’ before ’}’ token

[editing]

athena% gcc hello-world.c -o hello-world
athena% ./hello-world
Hello, world!
athena%
```

The C Preprocessor

The preprocessor is responsible for trimming your comments.

- Comments are understood to be between /* and */
- Comments are not between // and the end of the line
- Some compilers will support this latter comment style, but it can adversely affect the portability of your code

```
/* nothing in here is
 * going to be seen by the
 * compiler */

/* nor in here */
```

#include interprets the requested file

- Files between < and > will be sought amongst the system header files
- Files between " and " should be in the include path, which can passed to the compiler
- However, the include path by default will include the current directory

```
#include <stdlib.h>
#include "my-header.h"
```

#define defines macro substitutions

- These can be simpled 'defined' or 'not defined'
- Or they can be scalar values
- Alternatively, they can be functions with parameters
- These macro substitutions are recursively evaluated

```
#define _STRING_H
#define NULL (void *)0
#define SUM(a,b) ((a) + (b))
```

- Code between #if and #endif will be conditionally compiled
- #defined(SYMBOL) will evaluate true or false, depending on whether SYMBOL is defined or not
- The !, ||, and && operators work as expected
- Code to be skipped is replaced with blank lines
- Terminated with #endif

```
#if defined(MSDOS) || defined(OS2) || defined(WINDOWS)
#  if !defined(__GNUC__) && !defined(__FLAT__)

  /* conditionally compiled code goes here */

#  endif
#endif
```

#ifdef and #ifndef are convenient interfaces to common functionality:

- #ifdef SYMBOL is equivalent to #if defined(SYMBOL)
- #ifndef SYMBOL is similarly equivalent to #if !defined(SYMBOL)

```
#ifndef SYS16BIT
#  define SYS16BIT
#endif
```

#if 0 is a convenient way to comment out large swaths of code, particularly those that embedded comments. This latter point, because C comments are not recursive.

:-(

```
#if 0
#  include <sys/types.h> /* for off_t */
#  include <unistd.h>    /* for SEEK_* and off_t */
#  ifdef VMS
#    include <unixio.h>   /* for off_t */
#  endif
#  define z_off_t off_t
#endif
```

#pragma is used to use compiler implementation specific parameters and language extensions in a minimally standard way

```
#pragma warning(disable: 4035)  /* no return value */

#pragma map(deflateInit_,"DEIN")

#pragma message("LIBPNG reserved macros; \
use PNG_USER_PRIVATEBUILD instead.")
```

Language Structure

There are five different kinds of integer:

- char
- short
- int
- long
- long long

These can be either:

- signed (the default)
- unsigned

```
char i_8;                /* -128 to 127                  */
unsigned char ui_8;      /* 0 to 255                     */

short i_16;              /* -32768 to 32767              */
unsigned short ui_16;    /* 0 to 65536                   */

int i_32;                /* -2147483648 to 2147483647    */
unsigned int ui_32;      /* 0 to 4294967295U             */

long i_arch;             /* architecture
unsigned ui_arch;         * dependent                    */

long long i64;           /* -9223372036854775808L to
                          * 9223372036854775807LL        */
unsigned long long ui64; /* 0 to 18446744073709551615ULL */
```

There are two different kinds of floating point value:

- float
- double

```
float f_32;             /* roughly 3x10^-39 to
                         * roughly 3x10^39 */

double f_64;            /* roughly 1x10^-308 to
                         * roughly 1x10^308 */
```

Additionally, C understands strings and characters:

```
char zero = '0';

char *one_as_string = "One";

char *stuff = "I think I see "
              "Bob Marley "
              "in my cornflakes!\n";
```

Types can be operated on with:

- Arithmetic operators
- Bitwise operators
- Boolean operators
- Assignment operators

```
/* arithmetic operators:
 *   + - * /       : add, subtract, multiply, divide
 *   %             : mod
 *   ++a, --a      : increment/decrement and return
 *                 : new values
 *   a++, a--      : increment/decrement and return
 *                 : old values
 *
 * bitwise operators:
 *   & | ^ !       : and or xor not
 *   >> <<         : bitshift left and right
 *
 * boolean operators:
 *   > >=          : greater than, greater than or
 *                 : equal to
 *   < <=          : less than, less than or equal to
 *   == !=         : equal, not equal
 *   && || ^^      : and or xor
 *
 * assignment operators:
 *   =             : assignment
 *   += -= *= /=   : add, sub, mul, div and assign
 */
```

Flow control is achieved with:

- if contructs
- if/else if/else contructs

```
if(a > b) {
  /* only case */
}
if(a > b) {
  /* first case */
}
else if(b > c) {
  /* second case */
}
else {
  /* default case */
}
```

Some looping contructs are available:

- for
- while
- do ... while

The looping action can be regulated with:

- break
- continue

```
for(i=0;i<count;i++) {
  /* do something with i here */
  if(weird_case(i))
    continue;
}
while(predicate()) {
  /* do something here */
}
do {
  if(bored_p(i))
    break;
} while(predicate());
```

And the too often feared, and too often misunderstood, goto

```
/*
 * goto is great for getting out of nested loops
 * quickly and cleanly; other applications are
 * advised against
 */

for(y=0;y<height;y++) {
  for(x=0;x<width;x++) {

    if(super_badness_p(x,y))
      goto eject;

  }
}

eject:
```

The building block of computation in C is the function; all computation must occur inside one

```
/* start by declaring:
 *   - the return type
 *   - the function name
 *   - the types and names of the function parameters */

int sum_of_squares(int a,
                   int b)
{
  int c;                      /* first come variable
                               * declarations */

  c = (a * a) + (b * b);      /* then statements; each
                               * statement should end with
                               * a semicolon */

  return c;                   /* return a value, if we said
                               * we would */
}

int pythagorean_p(int a,
                  int b,
                  int c)
{
  double v1,v2;

  v1 = sum_of_squares(a,b);  /* calling functions occurs
                              * in the usual fashion */

  v2 = c * c;

  if(v1 == v2)
    return(1);
  else
    return(0);
}

/* main() is the entry point for program execution; define
 * it if you want your program to run */

int main()
{
  if(pythagorean_p(3,4,5))
    printf("3:4:5 is a pythagorean triple\n");
  else
    printf("3:4:5 is NOT a pythagorean triple\n");

  return(0);
}
```

Arrays, and Pointers

XXX arrays

The right side is empty.

XXX pointers

The right side is empty.

XXX arrays vs. pointers

The right side is empty.