C: Primer and Advanced Topics

Style

- Basics:
  - comments
  - white space
  - modularity
- Naming conventions:
  - variableNames ("*Hungarian Notation*": m_pMyInt, bDone)
  - FunctionNames
  - tTypeDefinitions
  - CONSTANTS

Brace Styles

- K&R:

```
if (total > 0) {
  printf( "Pay up!" );
  total = 0;
} else {
  printf( "Goodbye" );
}
```

- non-K&R:

```
if (total > 0)
{
  printf("Pay up!");
  total = 0 ;
}
else
{
   printf("Goodbye");
}
```


Variables and Storage

- Syntax:
  `<type> <varName> [= initValue];`
- Types (incomplete list):
  - char
  - short
  - int
  - long
  - float
  - double
  - all can be: signed (default) or unsigned

Operators

- Arithmetic Operators:
  `*, /, +, -, %`
- Relational Operators:
  `<, <=, >, >=, ==, !=`
- Assignment Operators:
  `=, +=, -=, *=, /=, ++, --`
  - don't abuse these, ie: `o = --o - o--;`
- Logic Operators:
  `&&, ||, !`
- Bitwise Operators:
  `&, |, ~, >>, <<`

Arrays

- Arrays start at **ZERO!** (a mistake you *will* make often, trust me)
- Arrays of int, float, etc. are pretty intuitive
  `int months[12];`
  `float scores[30];`
- Strings are arrays of char (C's treatment of strings is not so intuitive)
  - see Wang, Appendix 12 for string handling functions
- Multi-dimensional arrays:
  `int matrix[2][4];` (*not* `matrix[2,4]`)

Decision and Control

```
if( condition )
   statement;
else
   statement;
```

```
while( condition )
   statement
```

```
for( initial; condition; iteration )
   statement;
```

```
do
   statement;
while( condition )
```

- **break** and **continue** useful inside loops

Decision and Control (cont)

```
switch ( expression )
   case constant1:
          statement;
          break;
   case constant2:
          statement;
          break;
   default:
          statement;
          break;
```


Scope

- Scopes are delimited with curly braces
  "{" <scope> "}"
- New scopes can be added in existing scopes
- Child scopes inherit visibility from parent scope
- Parent scope cannot see into child scopes
- Outermost scopes are all functions
- *These scope rules are all similar to those of Turing and other common programming languages*

Functions

- Definition:
  <type> <functionName> ([type paramName], ...)
- No "procedures" in C … only functions
- Every function should have a prototype
- Example:

```
float area( float width, float height );

float area( float width, float height )
{
   return( width * height );
}
```


Preprocessor

#include (**<file.h>** versus **"file.h"**)

#define (constants as well as macros)

#ifdef (useful for debugging and multi-platform code)

```
   statements
#else
   statements
#endif
```


Structs

```
struct [<structureName>]
{
   <fieldType> <fieldName>;
} [<variableName>];
```

- structureName and variableName are optional, but should always have at least one, otherwise it's useless (can't ever be referenced)
- Example:

```
struct
{
   int quantity;
   char name[80];
} inventoryData;
```

Typedefs and Enumerated Types

`typedef <typeDeclaration>;`

- Example:

```
typedef int tBoolean;
tBoolean flag;
```

`enum <enumName> { tag1, tag2, ... } <variableName>`

- Example:

```
enum days { SUN, MON, TUE, WED, THU, FRI, SAT };
enum days today = MON;
```

or

```
typedef enum { SUN, MON, TUE } tDay;
tDay today = MON;
```


Pointers

- A pointer is a type that points to another type in memory
- Pointers are typed: a pointer to an int is different than a pointer to a long
- An asterisk before a variable name in its declaration makes it a pointer
  - *i.e.*: **`int *currPointer;`** (pointer to an integer)
  - *i.e.*: **`char *names[10];`** (an array of char pointers)
- An ampersand (&) gives the address of a pointer
  - *i.e.*: **`currPtr = &value;`** (makes currPtr point to value)
- An asterisk can also be used to de-reference a pointer
  - *i.e.*: **`currValue = *currPtr;`**

Pointers (cont)

- Use brackets to avoid confusion:
  - ie: **`*(currPtr++);`** is *very* different from **`(*currPtr)++;`**
- Using ++ on a pointer will increment the pointer's address by the size of the type pointed to
- You can use pointers as if they were arrays (in fact, arrays are implemented a pointers)

Command Line Arguments

```
int main( int argc, char *argv[] )
{
. . .
```

- **`argc`** is the number of arguments on the command line, including the program name
- The array **`argv`** contains the actual arguments
- Example:

```
if( argc == 3 )
   printf( "file1:%s file2:%s\n",
                          argv[1], argv[2] );
```


Casting

- You can force one type to be interpreted as another type through casting, ie:

```
someSignedInt = (signed int) someUnsignedInt;
```

- Be careful, as C has no type checking, so you can mess things up if you're not careful
- **`NULL`** pointer should always be cast, ie:
  - **`(char *) NULL`**, **`(int *) NULL`**, etc.

Library Functions for I/O

Opening and Closing Files (10.2)

```
FILE *fp;
fp = fopen( fileName, "r" );
fclose( fp );
```

- **fp** is of type "**FILE***" (defined in stdio.h)
- **fopen** returns a pointer (or NULL if unsuccessful) to the specified *fileName* with the given permissions:
  - "r" read
  - "w" write (create new, or wipe out existing fileName)
  - "a" append (create new, or append to existing fileName)
  - "r+" read and write

Character-by-Character I/O

```
fgetc( fp )  # returns next character from files referenced by fp
getc( fp )   # same as fgetc, but implemented as a macro
getchar()    # same as getc( stdin )
```

- These return the constant "**EOF**" when the end-of-file is reached

```
fputc( c, fp )  # outputs character c to file referenced by fp
putc( c, fp )   # same as fputc, but implemented as a macro
putchar( c )    # same as putc( c, stdout )
```


Line-by-Line Input

```
fgets( data, size, fp )  # read next line from fp (up to size)
gets( data )             # read next line from stdin
```

- **fgets()** is preferable to **gets()**
- Returns address of **data** array (or **NULL** if **EOF** or other error occurred)
- Example:

```
#define MAX_LENGTH 256
char inputData[MAX_LENGTH];
FILE *fp;
fp = fopen( argv[1], "r" );
fgets( inputData, MAX_LENGTH, fp );
```


Line-by-Line Output

```
fputs( data, fp )  # prints string "data" on stream referenced by fp
puts( data )       # same as fputs( data, stdout ) except a newline
                     is automatically appended
```


Formatted Output

```
printf( fmt, args ... )
fprintf( fp, fmt, args ... )
sprintf( string, fmt, args ... )
```

- Examples:

```
fprintf( stderr, "Can't open %s\n", argv[1] );
sprintf( fileName, "%s", argv[1] );
```

- **sprintf** example above better achieved with "**strcpy()**" function
- K&R book or man pages for all the details

Formatted Input

```
scanf( fmt, *args ... )
fscanf( fp, fmt, *args ... )
sscanf( string, fmt, *args ... )
```

- Examples:

```
fscanf( fp, "%s %s", firstName, lastname );
sscanf( argv[1], "%d %d", &int1, &int2 );
```

- Returns number of successful args matched … be careful, scanf should only be used in limited cases where exact format is know in advance
- See K&R book or man pages for all the details

Binary I/O

```
fread( buf, size, numItems, fp )
fwrite( buf, size, numItems, fp )
```

- Examples:

```
fread( readBuf, sizeof( char ), 80, stdin );
fwrite( writeBuf, sizeof(struct utmpx), 1, fp );
```

- Returns number of successful items read or written
- Other functions:

```
rewind(fp); fseek(fp, offset, kind); ftell(fp);
```

S-118