Lecture 8
C Programming Language

Variable Number Of Arguments

- How does printf work ?
  printf(const char * format, …);
  you can use 0 or more different variables instead of the ellipses(…)
- The first parameter must be explicit, so
  … can appear only at the end of the argument list.
- When this function is being called at run time, the number and type of arguments being passed must somehow be made known to the called function.
  In printf , the format string holds this information.
- How to refernce the unnamed arguments ?
  Functions in stdarg.h :
  - va_start: function to init access to args.
  - va_arg: function to access individual args.
  - va_end: function for clean up

Variable Number Of Arguments - cont.

-
  ```
  #include <stdarg.h>
  int sum(int argcnt, ….) /*argcnt is num of args */
  {
     va_list ap;           /* argument pointer (macro) */
     int ans = 0;

     va_start(ap,argcnt); /* init ap  */
     while (argcnt-- > 0) /* process all args */
       ans += va_arg(ap,int); /*va_arg advances ap */
     va_end(ap);            /* clean up */
     return(ans);
  }
  ```

- Use:
  ```
  int total = sum(5, 85, 90, 97, 79, 96);
  ```

Passing struct by value

-
  ```
  struct fraction {
          int number;
          int denom;
  };
  typedef struct fraction Fraction;

  void InitFraction(Fraction frac, int n, int d)
  {
          frac.number = n;
          frac.denom = d;
  }
  main()
  {
          Fraction f1;
          InitFraction(f1,1,2);
          printf("%d, %d\n,f1.number,f1.denom);
  }
  ```

Header Files - Review

- Declare in the header file any function accessible from another file that has a function prototype
- Declare in the header file any global variables accessible by a client. Use the extern modifier (when the variable is defined in another source file)
- do not put a definition - a declaration that allocates space - in a header file
- Include any #define constants to be used by the client in the header file
- Put macros to be used by the client in the header file
- Include data structure and typedef declarations used by the client in the header file

Makefile

- The UNIX make command follows a user-prepared description file known as Makefile, to perform its tasks.
- Structure of Makefile:

  ```
  target: zero or more components
  TABcommand1
  TABcommand2
  …
  ```

- Example - Dependencies rules:

  ```
  myprog : file1.o file2.o
  TAB  gcc file1.o file2.c  -o myprog
  file1.o : file1.c mydefs.h
  TAB gcc -c file1.c
  file2.o : file2.c
  TAB gcc -c file2.c
  ```

- Example: Default Dependencies:

  ```
  myprog : file1.o file2.o
  TAB  gcc file1.o file2.o -o myprog
  file1.o : mydefs.h
  ```

Makefile - cont.

- Macros: shorthand used in a Makefile

  ```
  NAME = value
  ```

- Example:

  ```
  CC = gcc
  OBJS = file1.o file2.o
  SRCDIR = user/aya/proj
  FLAGS = -g
  ```

- Using a macro: `$(NAME)`
- Example:

  ```
  myprog : $(OBJS)
                  $(CC) $(FLAGS) -o $@
  ```

- Example - multiple targets:

  ```
  all : p1 p2
  p1 : f1.o f2.o
          $(CC) $(FLAGS) -o $@
  p2 : f3.o
          $(CC) $(FLAGS) -o $@
  f1.o : f1.c mydefs.h
           $(CC) -c $(FLAGS) f1.c
  ```

Makefile - cont.

- The make command will perform the first task in the Makefile (all: in the last example)
- Additional maintenance tasks:

```
test : myprog
        rm -f test.out
        myprog  <test.in  >test.out

clean :
          rm -f $(OBJS)
          rm -f test.out core
```

- Use:

```
% make test
% make clean
```

System Calls

- The standard I/O routines are actually higher level functions that call low level UNIX **system calls**
- These system calls can be made directly for more low level programming

```
#include <sys/file.h>

int open(char *filename, int access, int mode);
int lseek(int fd, int offset, int origin);
int read(int fd, char *buffer, int k);
int write(int fd, char *buffer, int k);
mkdir (char *name, int mode);
int rmdir(char *dir_name);
chdir(char *dir_name);
```