Building a programming repertoire

Announcements

- Homework due Monday
- Will get considerably more difficult, so take advantage of relative free time
- Quiz today before break
- Don’t obsess over scanf

Another technique for passing data from the keyboard

- main() can also be written as

  main(int argc, char *argv[])
- If main is written in this way, information can be passed directly from the keyboard to the program at the time of execution
  - For example, we may run a program called a.out as: PROMPT > a.out Andrew Siegel
  - When a.out is run the two tokens Andrew and Siegel are passed into the program and can be obtained by querying argv and argc
  - Note: this involves some concepts that go beyond what we have learned so far. We will understand fully later.

Passing data, cont.

- When this technique is used, each token is stored as a separate *element* in the *array* argv
- The first token passed to the program is stored in argv[1], the second token in argv[2], etc.
- argc stores the (integer) number of tokens passed in
- A simple example will clarify this

argc/argv example

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

```
    printf("%s %d %s \n", "you entered", argc, "arguments");
    printf("%s: %s\n", "the zeroth arg is the program name", argv[0]);
     printf("%s: %s\n", "the first argument is", argv[1]);
    printf("%s: %s\n", "the second argument is, argv[2]);
}
```

```
> gcc argv_example.c –o argv_example
> argv_example hello world
  you entered 3 arguments
  the zeroth argument is the program name: argv_example
  the first argument is hello
  the second argument is world
```

argc/argv cont.

- Note that to do this completely generally we would need to use a *while* or *for loop*
- We will study while loops shortly
- Also note that argv reads all arguments as Strings (ie sequences of characters). So, we cannot simply pass two number and add them, for example. First, we would have to convert to a numeric type.

Converting String to integer

- An important function when using argv/argc is atoi.
- atoi converts a String argument to an integer in the following way:

```
int input, output;
input = atoi(argv[1]);
output = input + 1;
printf("%d\n", output);
> a.out 333
  334
```

Reading single characters

- Another important pair of functions for keyboard input/output is getchar/putchar
- getchar reads a single character from the *input stream*; putchar write a single character to the standard out for example:

```
int c;
c = getchar();  /* blocks until data is entered
                   if more than one char is entered only first is read */
putchar(c);     /* prints value of c to screen */
```

Getchar/putchar example

```
/* uses getchar with while to echo user input */
#include<stdio.h>
int main(){
  int c;              /* holds the input character value */
  c = getchar(); /* reads first character from input stream
                        with keyboard, this is signaled by Enter key*/
  while (1){      /* loop forever */
    putchar(c);  /* write char to keyboard */
    c = getchar(); /*get next char in stream */
  }
}
```

Input redirection

- Files can be sent to an input stream by using the unix redirection command '<'. For example, if we wish to pass input into a program call process_text, we can do:

  process_text < somefile.txt

  where somefile.txt is a text file that exists in the current directory. This sends the contents of somefile.txt into process_text as standard input

Simple C pre-processor directives

Using #define and #include

Variables vs. Constants

- We must distinguish between "variables" and "constants"
  - Variables: storage units whose contents can change over the course of a program
  - Constants: "hard-coded" expressions that are used directly, such as 32, 100.211, 'c', etc.
- It is bad programming style to have naked numbers (ie constants) appear frequently in a program – they are hard to change and their meaning is not clear
- By the same token, it is not always possible to use variables – compiler may need to know that value of a variable at compile time.

#define

- In C, constants can be defined with the #define *preprocessor directive*

```
#include<stdio.h>
#define MAX_ARGS 4
int main(int argc, char* argv[]){
  if (argc > MAX_ARGS){
     printf(%s %d\n,"Error, arg list exceeded",
    MAX_ARGS);
  /* etc, etc. */
 }
```

#define, cont.

- In the previous example, the preprocessor replaces all occurences of MAX_ARGS with the constant 4 before compiling
- This is preferable to using 4 directly, since changing it and understanding its role is much easier
- Note that by convention, constants have names with all uppercase letters
- Note that, in previous example, it is possible to use a variable called MAX_ARGS (or whatever). Why not do this?

putchar()/getchar() revisited

putchar()

- When putchar(***some_character***) is placed in a program, ***some_character*** is printed to standard out (screen by default).
- Note that ***some_character*** is just an integer type (char or int). When it is sent to putchar, the integer is converted to its ASCII character value and printed to the screen.
- Examples:

```
int x = 101; putchar(x); /* the letter e is printed */
int x = 'e'   ; putchar(x); /* same */
char x = 101; putchar(x); /* same */
char x = 'e';    putchar(x); /* same */
```

getchar()

- When used, returns the next character available from stdin.

  Example: Both snippets below store the next available char in the variable c. As long as the character code does not exceed CHAR_MAX, either is fine:

```
char c; c = getchar();
int c  ; c = getchar();
```

getchar(), cont.

- But what is meant by "the next available character from stdin"?
- Easiest way to see this is by *redirecting* a file into a program in Unix.
- Assume we have an executable named a.out and a file named file.txt. Then the command

  a.out < file.txt

  runs the program a.out sending the file file.txt to stdin. In other words, each time getchar() is called, the next character in the file will be returned.

Example: copying a file

```
/* copy.c -- copy input into output */
#include <stdio.h>
main(){
  int c; /* integer to hold each character read in */
  c = getchar(); /* read first character */
  while (c != EOF){    /* EOF is constant denoting file end */
     putchar(c); /* write c as char to stdout */
     c = getchar(); /read next character
  }
}
```

- This can be compiled and run as, for example:

```
> gcc copy.c -o copy
> copy < copy.c    #copy the source code itself
```

Redirecting stdout

- Just as stdin can be redirected from a file, so stdout can be redirected **to** a file.
- Example:

  PROMPT: copy < copy.c > copy.c.bak

  "Send the contents of file copy.c as stdin to the program copy, and send all stdout to the file copy.c.bak."

Example: Counting characters

```
/* count_chars.c */
#include <stdio.h>
main(){
  int num_chars = 0;  /* variable to hold number of chars */
  int c;         /* the current fetched char */
  c = getchar(); /* fetch the first char */
  while (c != EOF){ /* go until end of file is reached */
     ++ num_chars; /* same as num_chars = num_chars+1 */
      c = getchar(); /* fetch the next char */
   }
}
```

Note: There are a few things we can do to make this program more compact. See K&R page 18. Not really important at this stage, though.

Example: counting lines

```
#include<stdio.h>
main(){
  int num_lines = 0;
  int c;
  c = getchar();
  while (c != EOF){
     if (c = = '\n'){     /* check for newline characacter */
        ++num_lines; /* if so, increment line num counter */
     }
     c = getchar();
  }
}
```

Example: removing extra space

```
#include <stdio.h>
main(){
  int c, lastc;
  /* c: current character */
 /* lastc: previous character */
c = getchar(); /* get the first char in stream */
putchar(c);     /* write it out */
while (c != EOF){ /* until end of file */
   lastc = c;            /* store prev character */
   c = getchar();     /*  read next character in stream */
   if (c != ' '){       /*  if next character is not blank */
     putchar(c);      /*  write it to stdout */
   }
   if (c = = ' '){     /* if next character is a blank */
      if (lastc != ' '){   /* only write it if last character was NOT a blank */
         putchar(c);
      }
  }
```

Comments on getchar/putchar

- Note that when getchar() is called with no *return value*, ie simple:

  getchar();

  the next char is read, but it is not stored, any you have no access to it.
- EOF is a constant defined in <stdio.h>. It's value is returned by getchar() whenever an END-OF-FILE byte is encoutered.
- getchar() can take input directly from the keyboard as well as from a redirected file

One-dimensional arrays

- Allow the storage of multiple data items without having to use multiple unique variable names
- This is best seen with an example. Imagine that you would like to store the gpa of each student in the class. If there are 60 students, without using arrays this would require 60 variables, such as:

  float age1, age2, age3, age4, age5, …
- Arrays provide a convenient way to avoid this type of absurdity.

Arrays, cont.

- To store 60 student gpa's, we can create a single array variable as:

  float gpa_list[60];

- This declarations introduces an *array* of type float with 60 elements named gpa_list. Note that the name is up to the programmer. The type can be any of the types we've learned so far, and the *size* must follow the name surrounded by brackets. The size can be any integer > 0 so long as enough memory exists on the machine to accommodate.

Array assignment

- Once such an array is declared, we may assign to each of its *elements* as, for example:

  gpa_list[0] = 3.142;

  gpa_list[1] = 2.55; /* etc, etc */

- To access an array element is straighforward:

  gpa_list[5] = gpa_list[5] + .01; /* add .01 to 6th student gpa */

- Note that the in C, the elements are always assigned indices 0 --> size - 1.

Strings in C

K& R: Page 104, section 5.5

What is a String?

- String refers generically to a datatype for storing sequences of successive characters rather than single characters (char) or numbers (int, float, etc.)
- Typical examples:
  - Names
  - Lines of text
  - sentences

Strings as char arrays

- In C strings can roughly be derived from character arrays, as we saw in an early lecture with the reverse_lines.c example.
- Example:

```
char c, text[1000];
int j = 0;
while ( (c = getchar()) != EOF ){
  text[j++] = c;
}
```

Limitations of String as char arrays

- In preceding example, all information in input stream was stored in array *text*.
- However, a true String datatype should let us manipulate this data at a *higher level*. In other words, we would like to have commands that operate directly on strings, such as:
  - *Print entire string*
  - *Compare string to another string*
  - *Read entire string from file*

Limitations, cont.

- Of course, we can accomplish these tasks "by hand". That is, by writing a loop and printing each char, reading each char one by one, comparing each char, etc. Very cumbersome!
- Furthermore, imagine if we wanted to use this technique to create and store a string in a program. How would we initialize it?? Character by character would be horribly awkward:

```
char message[10];
message[0] = 'h'; message[1] = 'e';
message[2] = 'l';  message[3] = 'l';
```

Assigning to strings

- Fortunately, C provides higher-level techniques to accomplish these tasks.

1. Storing a string in a variable:
   - Example: char message[] = "Hello World";
   - Double-quotes must surround the text to be considered a string.
   - Char array syntax is used, but the size need not be specified by the programmer.

Printing strings

- Printing a string
  - Example: printf("%s", "Hello World");
  - Example:

    ```
    char message[] = "Hello World";
    printf("%s", message);
    ```

  - Note that %s is the string format specifier.
  - Programmer need not write a loop over each char. C provides direct capability of printing string

Reading strings

- Rather than reading a char at a time and pasting together a string, C can do this all at once with scanf (there are other ways we will discuss later).
  - Example:

    ```
    char name[100];
    scanf("%s", name);
    ```

  - Reads input directly into var name
  - Note that no & is needed when scanning into char array. We will understand why later.

String vs. char array

- Important to realize that a string is still very similar to a char array, but with some extra niceties for initialization.
- I say "very similar" because there is one important difference: A string is a char array terminated by the null character '\0'. This is how the various routines know when the end of the string has been reached so that it can be processed properly.

String manipulation Functions -- String.h library

- For now, I just expect you to be able to do the following:
  - Define/assign a string literal in a program
  - Print a string (we've been doing this for weeks).
  - Read a string from stdin using scanf.
  - Understand how a string is represented "under the hood".
- Next week we learn higher-level string manipulation functions -- fun.