Programming in C

Scott Schremmer
Outline

- Introduction
- Data Types and structures
- Pointers, arrays and dynamic memory allocation
- Functions and prototypes
- Input/output
- Comparisons
- Compiling/makefiles/debugging
Basic Data Types

- Variables must be declared before any instructions
- char, int, float, double
- no boolean!
- not initialized
- a string is represented as a character array
- char sampleString[20] = "sample string";
Structures

- No classes
- Create functions that work on structures of data
- Sample declaration:
  ```c
  struct person { int height, int weight}bob,sue;
  bob.height = 100;
  bob.height = 2*sue.height;
  ```
**Typedef**

* Used to refer to a type with a different name

```c
typedef unsigned int wholeNumber;
wholeNumber a,b,c;
a=5; /*etc*/
```

* typically used with struct
typedef struct optionalName {int height; int weight} person;

person bob, sue;

bob.height = 100; /*etc*/
Arrays

- int anArray[10];
- anArray is really a pointer to the beginning of an array
- no bounds checking or length available!
- anArray[20] may cause bizarre behavior
Pointers

The equivalent of an address:

```c
int *pointerToInt; int theInt;
pointerToInt=&theInt; /*"the address of theInt"*/
*pointerToInt = 5; /*follow the pointer*/
printf("%d",theInt);
```

5
Dynamic memory allocation

- int *anIntPtr, *anArray;
- anIntPtr = (int *)malloc(sizeof(int));
  - reserves space for 1 integer
- anArray = (int *)malloc(5 * sizeof(int));
  - reserves space for a 5 element array
- calloc--initialize memory to zero
Dynamic Memory Allocation

- int * anArray;
- anArray = (int *)malloc(10*sizeof(int));
- Equivalent:
  - anArray[3]=3;
  - *(anArray+3) =3;
- pointer arithmetic, increments by size of an integer
Dynamic Memory With Structs

- typedef struct {int height, int weight} person;
- person *bob;
- bob = (bob *) malloc(sizeof(person));
- equivalent:
  - (*bob).height = 5;
  - bob->height = 5;
- Pass to a function as a pointer
Creating a linked list

typedef struct listElem {
    Person *person;
    struct listElem *next;
} ListElem;

- head of list frequently a double pointer
- last element next=NULL
Dynamic memory allocation

- No garbage collection
- `free(aPointer);`
- Careful to free before all references are lost
- Free all elements of an array of pointers
- Memory leaks can be a significant problem
Don’t return Pointers to Local variables!

Don’t do:

```c
Person * someFunction() {
    Person *aPointerPerson;
    Person aPerson;
    aPointerPerson = &aPerson;
    return aPointerPerson;
}
```

Instead, malloc memory for new person
Functions

- int aSampleFunction(int a, int b)
  
  { return a*b;}

- Passed by value

- Except pointers, arrays

- Function prototype must exist prior to location in code
Sample function prototype

```c
int aSampleFunction(int, int)
main()
{
    printf("%d", aSampleFunction(5, 4));
}
int aSampleFunction(int a, int b)
{ return a*b;}
```
Prototypes and .h files

- Function may be in a separate file or library
- Link with after compiling
- Prototype usually contained in .h file
- `#include "file.h"` or `#include <file.h>`
- Put functions in file.c prototypes in file.h
- To include i/o functions:
  - `#include <stdio.h>`
Preprocessor DIRECTIVES

- Preprocessor run prior to compilation
- `#define CONSTANT value`
  - replaces CONSTANT with value (textual replace)
- `#define SUM(a,b) a+b`
  - macros, simply replaces SUM(this,that) with this+that
- use `-D` flag to set constants at compilation
- `#if,#else,#ifdef,#endif`
#include <stdio.h>

Output:

printf("formating string",arg1,arg2,etc);

special sequences: (man printf)

\n -- insert newline
\t -- insert tab
%d -- insert an integer value
%g -- insert a double
int anInteger = 5; int aDouble = 0.35;

printf("I am printing an integer %d\n and a double %g", anInt, aDouble);

I am printing an integer 5
and a double 0.35
To input from the standard input:

```c
int anInputInt;
scanf("%d", &anInputInt);
```

Note: need to pass a pointer to the `int`. 
FILE *filePointer;
filePointer = fopen("filename",mode);
/*Access the file*/
fclose(filePointer);

Sample modes: (man fopen)
- “r” text file for reading
- “w” text file for writing
- “a” append to existing text file
- “rb”, “wb”, “ra” as above with binary file
File I/O

- write to file:
  - fprintf(filePointer,"Astring",arg1,arg2,etc);

- read from a file
  - fscanf(filePointer,"Astring",arg1,arg2,etc);

- fgetc,fread
**Comparisons**

- No boolean types!
- `a<b`
  - returns 1 if `a<b`, 0 otherwise
- `while(1) {}`
  - infinite loop
- `if, while, do while, for` etc work as expected
Compilation

- Typically use gnu c compiler on linux machines
- gcc -o outfile file1.c file2.c file3.o
- gcc -c compiles only but doesn’t link (file.o)
- other options:
  - -w inhibit warning, -Wall include all warnings
  - -On (n=1,2,3) set optimization level
  - -g include debugging information
Makefiles

- Use dependencies to determine what to compile
- Good for large projects as limit code that must be recompiled
- Quirky about formatting, tabs important
Sample Makefile

all: client server

clean:
        rm client server

client: client.c
        gcc client.c -o client

server: server.c
        gcc server.c -o server
More Complex makefile

all: theProgram

theProgram: file1.o file2.o file3.o
    gcc file1.o file2.o file3.o -o theProgram

file1.o: file1.c headerFile.h
    gcc -c file1.c

file2.o etc....
Many more complex configurations possible

- variables for compiler and flags
- file lists to avoid an entry for each object file
DEBUGGER

- Call from command line
  - gdb executable
- debugger commands
  - run parameter list -- starts the program
  - setting break points
    - break file.c:10 -- break point on line 10
    - break function -- break point at start of function
Debugger

- Stepping through program
  - next -- executes current command (steps over functions)
  - step -- falls into functions
  - continue -- continue to execute until next breakpoint
- Displaying local variables
  - print variable_name
- where
  - indicates position in program and functions called
Debugger

- `bt`
  - back trace
- `disp variableName`
  - displays variable every time program pause
- `set variable variableName = 12`
  - used to modify variables
- `call function(arguments)`
  - immediately calls a function
  - can be used to display structures, lists etc.