C Review

CS 367 @ GMU

Programming Tools

- avoid the fancy IDE – go for command line tools!
  - gcc compiler
  - gdb debugger
  - text editors like emacs/vim

- work on your UNIX command familiarity

gcc

- gcc invokes C compiler; it translates C program into executable for some target
- Behavior controlled by command-line switches:

-o *file*	output file for object or executable
-Wall	all warnings
-c	compile single module (non-main)
-g	insert debugging code (gdb)
-p	insert profiling code
-l	library
-E	preprocessor output only

Using gcc

- Two-stage compilation
  - pre-process & compile: gcc –c hello.c
  - link: `gcc -o hello hello.o`
- Linking several modules:

  `gcc -c a.c` → a.o

  `gcc -c b.c` → b.o

  `gcc -o hello a.o b.o`
- Using math library
  - `gcc -o calc calc.c -lm`

Error reporting in gcc

- Common error sources
  - preprocessor: missing include files
  - parser: syntax errors
  - linker: missing libraries

C preprocessor

- The C preprocessor (cpp) is a macro-processor which
  - manages a collection of macro definitions
  - reads a C program and transforms it
  - Example:

```
#define MAXVALUE 100
#define check(x) ((x) < MAXVALUE)
if (check(i) { …}
```

becomes

```
if ((i) < 100)  {…}
```

C preprocessor

- Preprocessor directives start with # at beginning of line:
  - define new macros
  - input files with C code (typically, definitions)
  - conditionally compile parts of file

`#define` *name const-expression*

`#define` *name (param1,param2,…) expression*

- replaces name with constant or expression

- textual substitution
- symbolic names for global constants

Arithmetic Operators in C

Name	Operator	Example
Unary plus/minus	+/-	-7
Addition	+	num1 + num2
Subtraction	-	x - y
Multiplication	*	z * 6
Division	/	passengers / items
Modulus	%	m % n

Relational Operators

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	is equal to
!=	is not equal to

Relational expressions evaluate to the integer

values 1 (true) or 0 (false). They are all *binary* operators.

Logical Operators

- Logical operators are used to combine simple conditions to make complex conditions.

&&	is AND	if (x > 5 && y < 6)
||	is OR	if (z == 0 || x > 10)
!	is NOT	if (! (b > 42))

Pointers and Arrays

Pointers

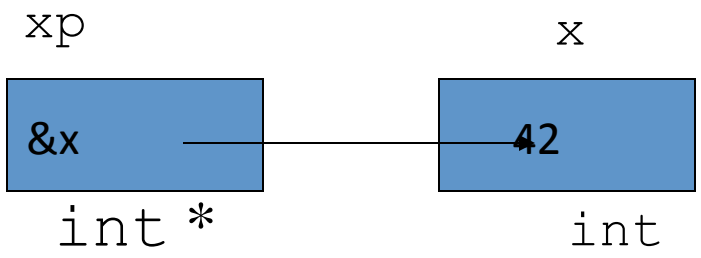
- Can think of pointer as a type, like int, float, etc.
- Pointer holds location of variable in memory (usually a memory address)
- Operations on pointer variable allow program to access data at that location

Data objects and pointers

- The memory address of a data object, e.g., `int x`
  - can be obtained via `&x` (`&` = the address operator)
  - has a data type `int *` (in general, *`type` `*`*)
  - has a value which is a large (4/8 byte) unsigned integer
  - can have pointers to pointers: `int **`
- The size of a data object, e.g., `int x`
  - can be obtained via sizeof x or sizeof(x)
  - has data type `size_t`, but is often assigned to `int` (bad!)
  - has a value which is a small(ish) integer
  - is measured in bytes

Data objects and pointers

- Every data type T in C has an associated pointer type: T *
- A value of type * is the address of an object of type T
- If an object `int *xp` has value `&x`, the expression `*xp` "dereferences" the pointer and refers to `x`, thus has type `int`



Pointers

- Allows us to indirectly access variables

  in other words, we can talk about its *address* rather than its *value*

```
 int a=42, b=16;   /* allocate space for 2 ints */
   int *p, *q;  /* allocate space for 2 memory
                   addresses that can hold address
                   of integers */
   char *r;       /* allocate space for memory address
                   that can hold address of a
                   character. */
```

Parameter passing

- In C, by default, parameters are passed to the functions *by value*
- Changes made to the parameter during the execution of the function do not affect the value of the argument in the calling function
- What if we want to have the function change the value of the parameter?

Pointers as reference parameters

```
void swap(int *x, int *y) {
    int t;
    t = *x;
    *x = *y;
    *y = t;
}

int a=42, b=16;
swap(&a,&b);
```

Arrays

- A list of values arranged sequentially in memory
- Expression `a[4]` refers to the 5th element of the array `a`
- Arrays are defined by specifying an element type and number of elements
  - `int vec[100];`
  - `char str[30];`
  - `float m[10][10];`
- For an array containing *N* elements, indexes are 0..*N*-1
- Stored as linear arrangement of elements

Common Pitfalls with Arrays in C

- Overrun array limits
  - There is no checking at run-time or compile-time to see whether reference is within array bounds.
-
```
int array[10];
int i;
for (i = 0; i <= 10; i++) array[i] = 0;
```

- Declaration with variable size
  - Size of array must be known at compile time.
-
```
void SomeFunction(int num_elements) {
  int temp[num_elements];
  …
}
```

Passing Arrays as Arguments

- C passes arrays by reference
  - the address of the array (i.e., of the first element)
    is written to the function's activation record
  - otherwise, would have to copy each element

This must be a constant, e.g.,

```
#define MAX_NUMS 10
```

```
main() {
   int numbers[MAX_NUMS];
   mean = Average(numbers);
}
int Average(int inputValues[MAX_NUMS]) {

   for (index = 0; index < MAX_NUMS; index++)
    sum = sum + indexValues[index];
   return (sum / MAX_NUMS);
}
```

Passing Arrays as Arguments

- When the parameter in a function definition is a one-dimensional array, the length of the array does not need to be specified:

```
• void some_function(int a[])
{
..
}
```

- In this case, if the function needs to know also the size of the array, the size of the array can be passed as an additional parameter.
- Note: When a parameter is a multidimensional array, only the length of the first dimension may be omitted in the function definition → `void some_function(int a[][LEN])`

Relationship between Arrays and Pointers

- An array name is essentially a pointer to the first element in the array

```
char word[10];
char *cptr;

cptr = word;  /* points to word[0] */
```

- *Difference:*
  Can change the contents of cptr, as in
- `cptr = cptr + 1;`
- (But: the identifier "`word`" is not a variable.)

Pointer Arithmetic

- Address calculations depend on size of elements
- C does size calculations under the covers, depending on size of item being pointed to:
- `double x[10];`
- `double *y = x;`
  `*(y + 3) = 13;`

allocates 20 words (80 bytes)

same as x[3] -- base address plus 6

- When two pointers are subtracted, the result is the distance (measured in array elements) between the pointers.

```
p = &a[7]; q = &a[2]; i = p - q; /* i is
5 */
```

Pointer Arithmetic

- Arithmetic operations on a pointer x is meaningful only when x points to an array element. Also, subtracting two pointers gives a meaningful result only when both point to elements of the same array.

Combining the * and ++ operators

- Example: update the value of an array element, and increment the index (advance to the next element): `a[i++] = j;`
- If `p` is a pointer to a[i] (i.e., the address of a[i]), this can be written as: `*p++ = j;`
- This is equivalent to: `*(p++) = j;`
- Here p will be incremented after the assignment!

More on arrays and pointers

- `int a [10];`

- `*a = 15;/*equivalent to a[0] = 15; */`
- `*(a+1) = 9 /* equivalent to a[1] = 9; */`

- Remember:
- `a + i` is the same as `&a[i]`
- `* (a + i)` is the same as `a[i]`

More on arrays and pointers

- Simple code to sum up elements of an array:

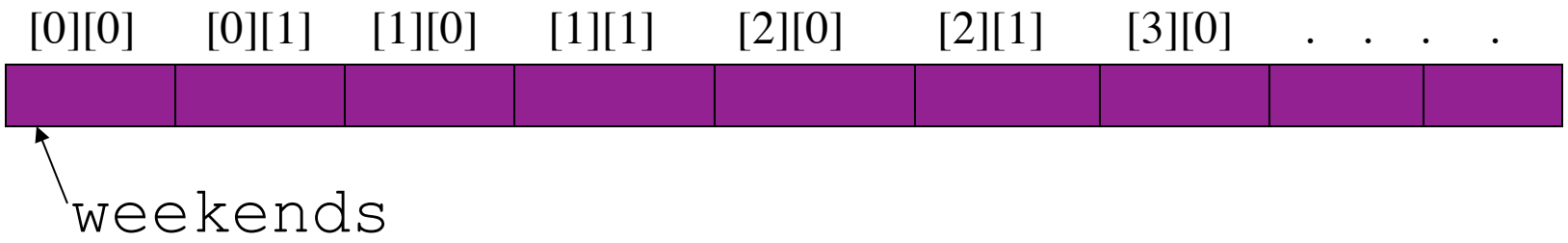
```
#define N 100

int a[N], sum, *p;

sum = 0;
for (p = a; p < a + N; p++)
    sum += *p;
```

2-Dimensional Arrays

```
int weekends[52][2];
```



- `weekends[2][1]` is same as `*(weekends+2*2+1)`
  – NOT `*weekends+2*2+1` :this is an int !

I/O

- standardized functions call UNIX I/O functions
- FILE type abstracts a text stream: actual text files, stdin, stdout, stderr, etc.

```
FILE *someFile = fopen("thisfile.txt","w");
fclose(someFile);
```

- read/write text lines: `fgets()`, `fputs()`
- formatted r/w: `fscanf()`, `fprintf()`
  - use formatting characters: `%d`, `%f`, `%x`, `%c`, `%s`, …
- stdin/stdout versions:
  - `putchar()`, `getchar()`, `printf()`, `scanf()`

debugging – print statements

- quick and easy
- have to dig through outputs
- no interaction/changes while program is run
- might affect program behavior!

debugging – source level (gdb)

- step by step (in/over/out)
- set breakpoints, watchpoints
- monitor watchlists
- modify memory on the fly, see effects

multi-file programs

- include other files (think: "paste it here")

```
#include <stdio.h>      /*standard imports*/
#include "mycode.h"     /*user code imports*/
```

- Makefiles: recipe for compiling multiple files
  - set dependencies (check freshness by timestamps)
  - give per-file build commands

```
thisfile.o: thisfile.c relies.h on.h these.h files.h
    gcc -g -c thisfile.c
```