CSE320 System Fundamentals II

TONY MIONE

Topics

Bit manipulation

Pointers in Depth

Functions in Depth

C – A ‘Systems’ Language

Pointers provide easy access to system structures [i.e. heap/memory allocation structures, process tables, etc]

Low-level bit operations provide for manipulation of flags in data structures

Loops allow easy operation repetition (easier than assembler)

Bit manipulation

C provides *bitwise logical* operators. Useful for:

- Isolating bits or bit fields
- setting bits
- Unsetting (turning off) bits

C provides *shift* operators that help create *masks*

Accessing Bit Fields

Bit Fields –

- Collection of contiguous bits
- non-byte-aligned
- field size not likely an integral number of bytes

Extracting value is easy (relatively) with C operators

Setting a value is relatively easy

Accessing Bit Fields [Example]

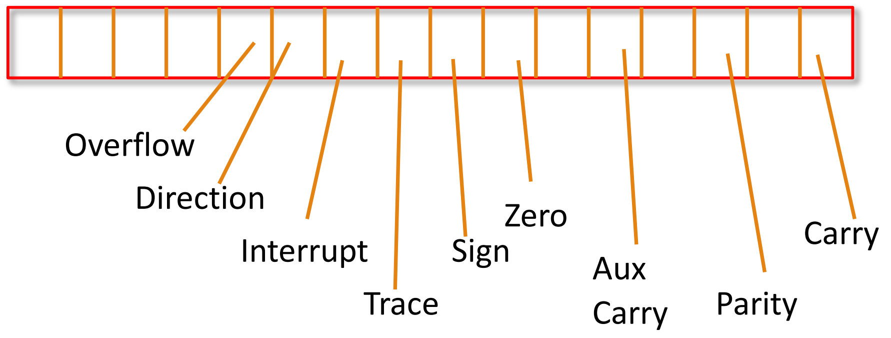
Example: Bit 27 of a fictitious processor's status register indicates if the last operation was negative (1=true). Extract the bit and test it

```
  const int negShift = 27;
  int negMask = 1<<negShift;        // This creates a mask of 0x08000000
  if ((psr&negMask)>>negShift) {  // This isolates bit 27 and shifts it to the low-
order bit
    printf("PSR: Neg is on\n");
  } else {
    printf("PSR: Neg is off\n");
  }
```

Setting Bit Fields [Example]

Example: The top 5 bits of a 32 bit value must be set to 28 decimal (0x11100). This must overwrite the old value in those 5 bits:

Exercise



Above is a status word for early 80x86 processors. Each box represents a single bit [0/1]. Bits like Overflow, Sign, Zero, and Carry are set after any arithmetic instruction and can be tested by moving. Assuming there were C code to fetch the PSR value and place it in a 'short' variable in C, write a function to check the sign bit and indicate if the result was negative (sign bit is 1). Also, write a function to check the zero bit to check if the result was zero (zero bit is 1).

The prototypes for the functions should be:

```
int resultNegative(short psr);
int resultZero(short psr);
```

Write a main() to call the functions with the following PSR values: 0x00C1, 0x0045, 0x0011

Pointers in Depth

Pointers – Description / Use

Pointer Syntax

Pointer Examples

Pointers – Description / Use

Pointers hold the address of a data item

- Size of a pointer variable is the size of an architecture's address
  - 32-bit architectures – pointer variable is 4 bytes
  - 64-bit architecture – pointer variable is 8 bytes
- Must 'dereference' the variable with '*' to get actual data

Syntax may be confusing between declarations and use

Uses:

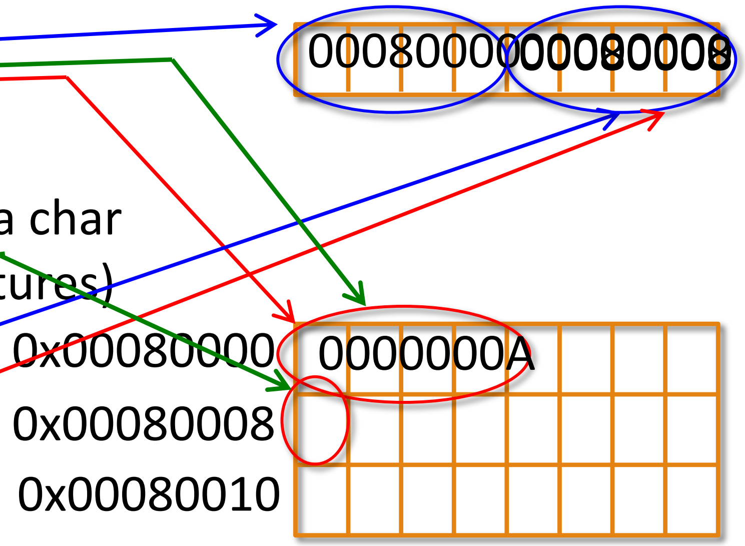
- Passing large data between functions
- Returning multiple values from functions
- Notational convenience – interchangeable with array variables

Pointer Definitions

Use ‘*’ in front of variable name – ‘*’ is descriptive indicating the variable name points to the named type

// pointerToMyInteger is a memory address of an int
// size of pointerToMyInteger is 4 bytes (on 32 bit architecture)
// size of *pointerToMyInteger is 4 bytes (when compiler considers ‘int’s to be 32-bits)
int *pointerToMyInteger = 10;

// pMyChar is will hold the memory address of a char
// size of pMyChar is 4 bytes (on 32 bit architecture)
// size of *pMyChar is 1 byte
char *pMyChar;
pMyChar = (char *) malloc(sizeof(char));



Pointer Definitions

```
struct _tempStruct {
   int amTemp;  // Temperature at 8 AM
   int pmTemp; // Temperature at 6 PM
   char location[20]; // Holds a city name
}

// pTempStruct is a memory address of an _tempStruct
// size of pTempStruct  is 4 bytes (on 32 bit architecture)
// size of *pTempStruct is 28 bytes (when compiler considers ‘int’s to be 32-bits).
// Also, cannot reference *pTempStruct as a whole. Must refer to fields within
struct
struct _tempStruct *pTempStruct;
```

Pointer Definitions with Initialization

Can initialize variables

```
// Memory is reserved and the value 5 is placed there.
// The address returned from malloc() is placed in the variable pointerToMyInteger.
// The memory allocated by malloc is uninitialized.
int *pointerToMyInteger = malloc(sizeof(int));

// A variable is created called anotherMyInteger and the value 5 is stored there
int anotherMyInteger = 5;

// A pointer variable is created and initialized to point at the memory from
anotherMyInteger
int *pointerToAnotherInt = &anotherMyInteger;
```

Pointer Use

Setting a pointer variable

Referencing ‘value’

Passing pointers to function (possibly for returned data)

Stepping through arrays

Pointer Use

Setting a pointer variable involves generating the ‘address’ of a another variable

‘&’ asks compiler to generate ‘address’ of the variable name

‘*’ references the contents of the memory to which a variable points.

- In executable code, ‘*’ describes an action to access memory

```
int *pMyInt;
int oneValue, userValue = 10;   // Create two ints. Set userValue to 10
…
pMyInt = &userValue;  // pMyInt points to the storage where userValue was created
oneValue = *userValue; // Copies the memory contents of userValue (10) into one Value
```

Pointer Use

Some functions return multiple values

```
// readline
//  fileDesc – File descriptor of an open file
//   buffer – (output) Holds pointer to a buffer allocated (by readline()) for returned file
data
//   length – (output) Number of characters read by function
int readline(int fileDesc, char **buffer, int *length) {…}

…
char *buffer;
int status, length;
int fd = open(“File.txt”, ‘r’);
…
status =readline(fd, &buffer, &length);
```

Pointer Use

If pointers refer to an array element, they can be used to iterate through array

- ++, -- operators will increment/decrement pointer by the size of the element to which it points (i.e. 4 bytes for int *)

```
int myArray[] = {5, 10, 15, 20};
int *arrPtr = myArray;
int idx;

for (idx = 0; idx < sizeof(myArray)/sizeof(int); idx++)
{
   printf("Value: %d\n", *arrPtr);
   arrPtr++;
}
```

Exercise

Write a simple for() or while() loop to step through the array using the provided pointer and print each value up until the value is 9. You can separate the printed values with newlines or with spaces.

```
#include <stdio.h>
#include <unistd.h>

void prary(int *aryptr) {
   // Write a loop to print values until the integer at the location is ‘9’
}

int main (int argc, char **argv) {
   int theValues[10] = {1,2,3,4,5,6,7,8,9,0};
   int *ptr = theValues;
   prary(ptr);
}
```

Functions in Depth

Function declarations (prototypes) vs definitions

Arguments

Inlining functions

Functions Declarations vs. Definitions

Functions must be declared before being called

Declarations can go in a ‘header’ file

- Show return type
- Show arguments and types
- No body

Fully defined function in a ‘.c’ source file contains code

Function Declarations vs. Definitions

Declarations (prototypes in header file):

```
double computeSquare(double num);
int open(char *name, char *mode);
```

Note:
Semicolon at end of line.
No braces.

Definitions (function code in .c file):

```
double computeSquare(double num) {
    return num * num;
}
int open(char *name, char *mode) {
   …
}
```

Note:
No semicolon at end of line, braces, and complete code.

Arguments

C uses **Call-by-value**

- **Call-by-value** – The value of variables are passed to functions
- **Call-by-reference** – The address of (or reference to) a variable is passed to a called function

Call-by-reference can be simulated using pointer variables

Simulation

Call-by-value vs Call-by-reference

Examples

Using regular variables:

```
#include <math.h>
void computeHypot(double a, double b, double hyp) {
  // This computes a result but the new value does not make it back
  // to the caller!!!
  hyp = sqrt((a * a) + (b * b));
}
double computeSquare(int num) {
  return num * num;
}
…

double result;
computeHypot(5, 6, result);
```

Examples

Using pointers (call-by-reference):

```
#include <math.h>
void computeHypot(double a, double b, double *hyp) {
   // This computes a result but the new value does not make it back
   // to the caller!!!
  *hyp = sqrt((a * a) + (b * b));
}
…

double result;
computeHypot(5, 6, &result);
```

Function Inlining

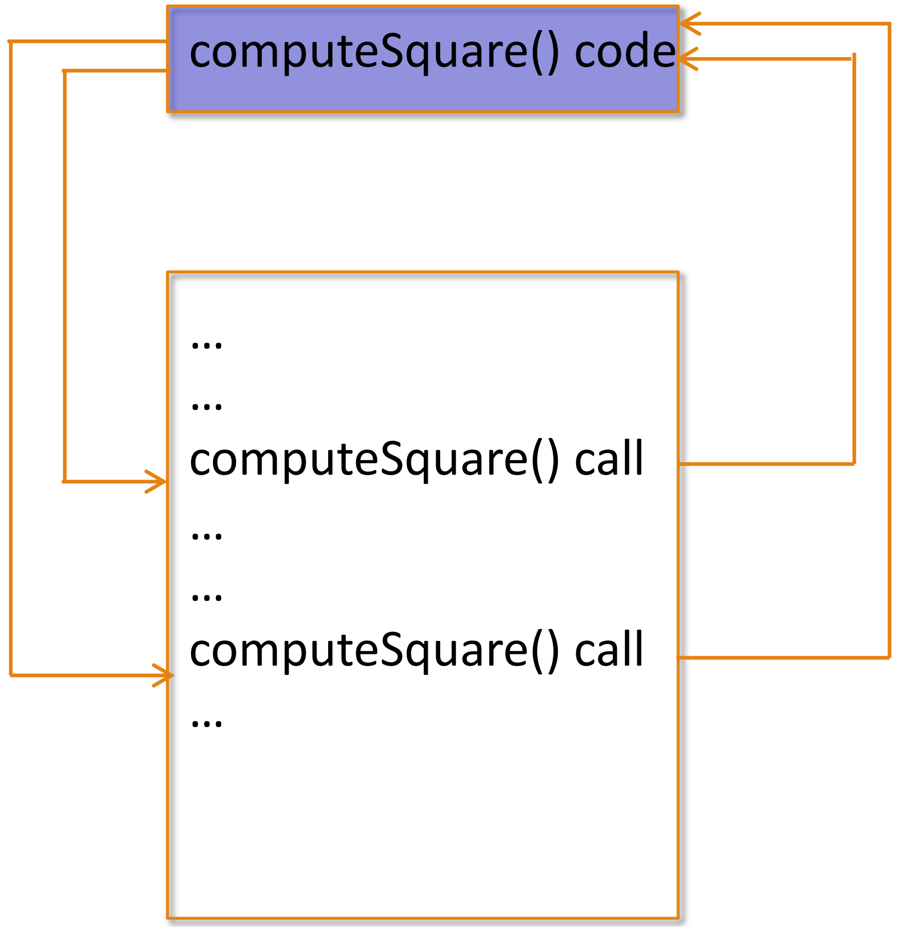
Functions add overhead to code:

- Setting up arguments
- Transferring control to function
- Setting up return argument
- Transferring control back to caller

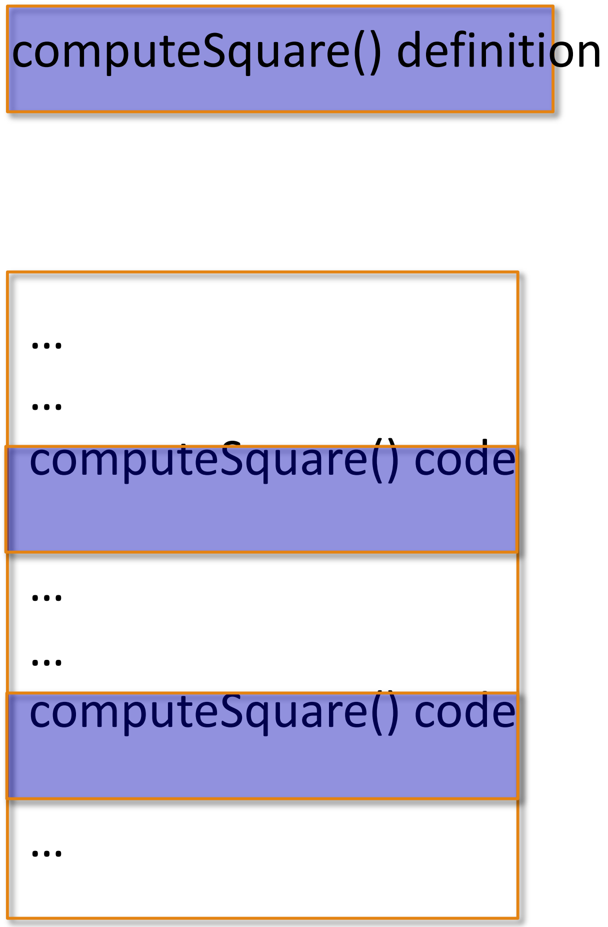
For efficiency, use *inline* with small functions

- *Inline* causes a copy of the function code to be expanded at each call
- Costs memory for extra copies of code
- Saves run time during execution
- The *inline* keyword is a **hint** to the compiler

Function Calls (no inlining)



Function Inlining



Function Inlining

No inlining:

```
double computeSquare(double num) {
  return num * num;
}
…
int sqr = computeSquare(5);
```

inlined:

```
inline double computeSquare(double num) {
  return num * num;
}
…
int sqr = computeSquare(5);
```

Note 1:
Only difference is *inline* keyword ahead of declaration.

Note 2:
Function call does NOT change!

Function Inlining

No inlining:

```
double computeSquare(double num) {
  return num * num;
}
…
int sqr = computeSquare(5);
```

Forced inlined:

```
__attribute__((always_inline))
…
inline double computeSquare(double num) {
  return num * num;
}
…
int sqr = computeSquare(5);
```

Note 1:
To force the compiler to inline, specify the compiler directive listed here [only needed once/file.]

Exercise

Try to force the C compiler to inline a function. Place code below in randPlusFive.c

- Build the code with switches '-O1 –S –c randPlusFive.s'.
- Add code and directives to tell gcc to inline the function and save to randPlusFiveInline.c
- Build the new code with switches '-O4 –S –c randPlusFiveInline.s'

```
#include <stdio.h>
#include <stdlib.h>

int plusFive(int startvar) {
  return startvar + 5;
}
int main (int argc, char **argv) {
  int result;
  int randnum = rand();

  result = plusFive(randnum);
  printf ("randnum = %x\nresult = %x\n", (int) (randnum), (int) (result));
}
```

Compare the two .s files to see differences in generated assembler.

Compiling C programs under Linux

GNU C (gcc) is used to compile C programs

Syntax:

- gcc –o <executableFileName> <sourceFileName>.c

Compiling Multi-Module C programs

Larger applications may be split between multiple source files

- Use several gcc commands
- Commands must indicate ‘compile only’ (-c)

Syntax for each source file:

- gcc –c –o <relocatableFileName>.o <sourceFileName>.c

Link command:

- gcc –o <executableFileName> <list of .o files>

Specifying Libraries when Linking

Some applications need system libraries

- Libraries are in system directories (/usr/lib, /usr/local/lib)
- File names of form: lib<nm>.so (i.e. libm.so is ‘math’ library)

Link command:

```
gcc –o <executableFileName> <list of .o files> -l<nm> -l<nm> …
```

Example:

```
gcc –o bigprog bigprocmain.o bigprocutils.o -lm
```

Resources

http://www.cprogramming.com/

Questions?