Memory-Related Perils and Pitfalls

- Using * with ++
- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

Memory-Using * and ++ together ptrplusplus.c

- Avoid using * and ++ in the same expression.
- What's the difference among

```
*ptr++
(*ptr)++
++*ptr
++(*ptr)
```

- `*ptr++` dereferences ptr, then increments ptr
- `(*ptr)++` performs a post-increment on what ptr points to
- `++*ptr` performs a pre-increment on what ptr points to
- `++(*ptr)` performs a pre-increment on what ptr points to

Memory-Dereferencing Bad Pointers

- The classic scanf bug is to pass variable itself instead of an address
- Typically reported as an error by the compiler.

```
int val;

...

scanf("%d", val);
```

Memory-Reading Uninitialized Memory

- Assuming that heap data is initialized to zero is wrong, see calloc if needed.

```
/* return y = A times x */
int *matvec(int A[N][N], int x[N]) {
   int *y = malloc( N * sizeof(int));
   int i, j;

   for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
         y[i] += A[i][j] * x[j];
   return y;
}
```

Memory-Overwriting Memory

- Allocating the (possibly) wrong sized object
- Below, the second line should have been sizeof(int *).

```
int i, **p;

p = malloc(N * sizeof(int));

for (i = 0; i < N; i++) {
   p[ i ] = malloc(M * sizeof(int));
}
```

- Not checking the max string size

```
char s[8];
int i;

gets(s);  /* reads "123456789" from
stdin */
```

- Basis for classic buffer overflow attacks
  - 1988 Internet worm
  - Modern attacks on Web servers
  - AOL/Microsoft IM war

Memory-Overwriting Memory (pt 2)

- Misunderstanding pointer arithmetic:
  - Remember: p+=N already adds N times the sizeof (int 8) to p

```
int *search(int *p, int val) {

   while (*p != NULL && *p != val)
      p += sizeof(int);

   return p;
}
```

Memory-Referencing Nonexistent Variables

- Another error that is commonly seen is returning a pointer to a local variable (the variable's lifetime ceases at the end of the function and pointer is not safe to use.)
- Don't forget that local variables disappear when a function returns

```
int * sum(int a, int b){
  int c = a + b;
  return &c;
}
```

Memory-Freeing Blocks Multiple Times

- Overkill…

```
x = malloc(N * sizeof(int));
        <manipulate x>
free(x);

y = malloc( M * sizeof(int));
        <manipulate y>
free(x);
free(y);
```

Memory-Referencing Freed Blocks

- Evil!!!

```
x = malloc(N * sizeof(int));
  <manipulate x>
free(x);
   ...
y = malloc(M * sizeof(int));
for (i = 0; i < M; i++)
   y[ i ] = x[ i ]++;
```

Memory-Tip for both of the 'Freeing Scenarios above:

- Considering setting pointers to NULL after deallocation as a bookkeeping measure. Later, you can check if a pointer is NULL before using it. Deallocating a NULL pointer has no effect.

(on the other hand…) Memory-Failing to Free Blocks - Memory Leaks

- Slow, long-term killer!
- Here, a function allocated memory and tracked it with a pointer which doesn't exist after the function. The memory is no longer tracked by the program but is left allocated. Over time the system may run out of memory.

```
int foo(int x,int y, int z) {
    int result;
    int *p = malloc(n * sizeof(int));
    …
    //forget to free p
    return result;
}
```

Memory-Failing to Free Blocks - Memory Leaks (cont.)

- Freeing only part of a data structure
- There are multiple problems here. Freeing head only deallocated a small block of memory consisting of two pointers. The nameString block was not deallocated nor was the rest of the linked list structure.

```
typedef struct list {
   char * nameString;
   int val;
   struct list *next;
} LIST_t;
foo() {
   struct list *head = malloc(sizeof(LIST_t));
   head->val = malloc((NAMESIZE+1)*sizeof(char));
   head->next = NULL;
   <create and manipulate the rest of the list>
    ...
   free(head); //only use of free
   return;
}
```

Memory-Dealing With Memory Bugs

- Conventional debugger (gdb)
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs
- Some malloc implementations contain checking code
  - Linux glibc malloc: setenv MALLOC_CHECK_ 2
- Binary translator: valgrind (Linux)
  - Powerful debugging and analysis technique
  - Rewrites text section of executable object file
  - Can detect all errors as debugging malloc
  - Can also check each individual reference at runtime
    - Bad pointers
    - Overwriting
    - Referencing outside of allocated block
- Garbage collection (Boehm-Weiser Conservative GC)
  - Let the system free blocks instead of the programmer.

Interrupts-Dealing With Memory Bugs

- Reading: Book 7.6
- Also: http://www.scriptoriumdesigns.com/embedded/interrupts.php

Interrupts-References and/or Reading Resources

- Introduction to Embedded Programming ASM and C examples
  - http://www.scriptoriumdesigns.com/embedded/interrupts.php
  - http://www.scriptoriumdesigns.com/embedded/interrupt_examples.php interrupt examples
- GNU C Programming:
  - http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html
- Newbie's Guide to AVR Interrupts
  - http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=89843
- Using Interrupts
  - http://www.pjrc.com/teensy/interrupts.html
- Interrupt driven USARTs
  - http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=48188>
- Atmega169P Datasheet
  - http://www.atmel.com/dyn/resources/prod_documents/doc8018.pdf
- Beginners Programming in AVR Assembler
  - http://www.avr-asm-tutorial.net/avr_en/beginner/RIGHT.html
- Code segments cseg and org
  - http://www.avr-asm-tutorial.net/avr_en/beginner/JUMP.html
- A Brief Tutorial on Programming the AVR without Arduino
  - https://www.mainframe.cx/~ckuethe/avr-c-tutorial/

Interrupts-What is an interrupt

- What is an Interrupt?
  - An interrupt is a signal (an "interrupt request") generated by some event external to the CPU , which causes the CPU to stop what it is doing (stop executing the code it is currently running) and jump to a separate piece of code designed by the programmer to deal with the event which generated the interrupt request.This interrupt handling code is often called an ISR (interrupt service routine).When the ISR is finished, it returns to the code that was running prior to the interrupt...
- Pasted from http://www.scriptoriumdesigns.com/embedded/interrupts.php

Interrupts-Overview

- Interrupt Sequence

  1) Interrupt Event ->

  2) generates interrupt request ->

  3) triggers interrupt service routine

  - A varying number of ISRs may be supported, typically in range 1-16
  - Multiple events may be designated (mapped) to a single routine
  - Interrupt events and may have priorities
  - An ISR is implemented inside a function with no parameters and no return value (void)
  - Typically you should keep interrupt routines shorter than 15-20 lines of code (why do you think?)

Interrupts-Sources

- Two main sources of interrupts
  - **Hardware Interrupts** are commonly used to interact with external devices or peripherals. In the case of microcontrollers the peripherals may be on-chip.
  - **Software Interrupts** are triggered by software commands, usually for special operating system tasks like switching between user and kernel space, or handling exceptions (not provided directly in AVR)

- Some common hardware interrupt sources:
  - Input pin change (Pin Change Interrupt)
  - Hardware Timer overflow or compare-match (timer reaches a specified value)
  - Peripherals for Common Serial Communication
    - UART, SPI , I2C
      - RX data ready
      - TX ready
      - TX complete
  - ADC conversion complete
  - Watchdog timer timeout

Interrupts-Advantage Over Software Polling

- Interrupts avoid writing software code in such a way that it (the processor) must frequently spend time checking the status of the input pins or some registers...instead let custom hardware do that.
- In Polling there is busy-wait.
- Busy-Wait: the CPU is busy by executing instructions, but it is just waiting for an event to happen:

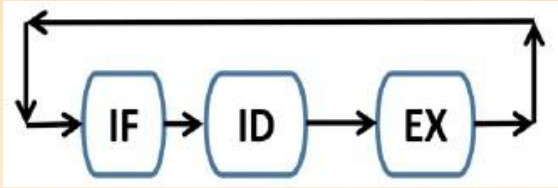
```
void USART_Transmit ( unsigned char data ) {
  while ( !( UCSR0A & ( 1 << UDRE0)) );
  UDR0 = data;
}
```

Busy-wait

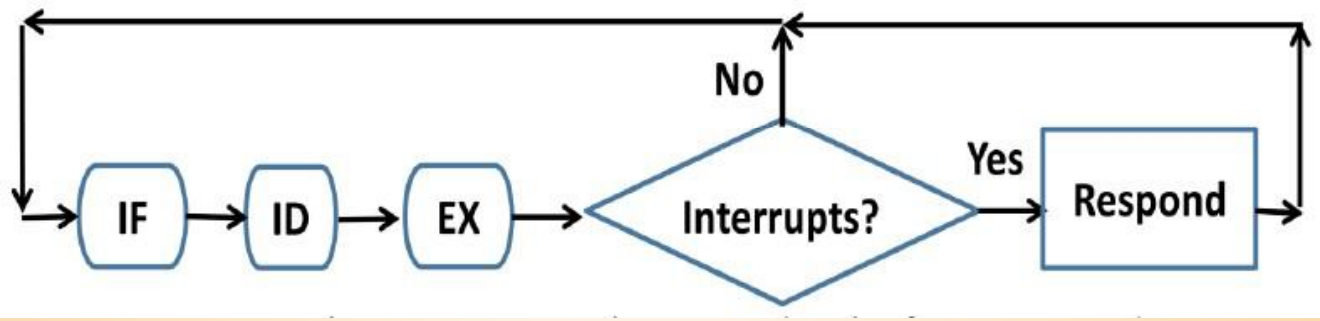
- By Interrupts, hardware checks to see if an event has occurred or not. (the CPU does not wait for the event to occur, but before executing each instruction, it checks to see if it has happened or not – see next slide:)

Interrupts-Interrupt Service Flow:

- No Interrupt Check



- Interrupt Check



  - After executing each instruction, the CPU checks for any pending interrupts.
  - If there is an interrupt, the CPU transfers control by saving the PC (Program Counter) and loading the address of an ISR (Interrupt Service Routine) into the PC.
    - Each interrupt has its own address in the interrupt vector, (more on this later)
  - After handling the interrupt, the old PC values are restored, and execution of the previous program resumes.

Interrupts-Vocabulary and Concepts

- **Interrupt Event** generates an **interrupt request** (IRQ). The system responds by interrupting the code flow and "vectoring" off to an **interrupt service routine** (ISR)

- A service routing is implemented much like a regular function except it has no parameters (void) or return type (void).

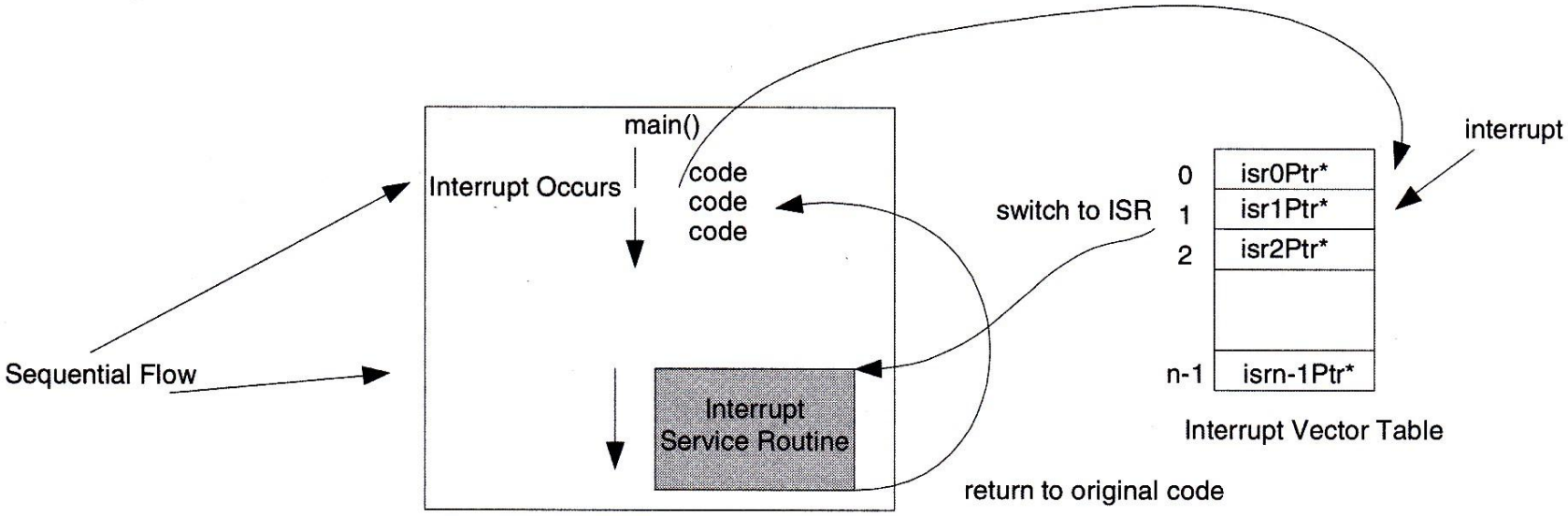


Figure 7.45 Following an Interrupt