TDDI11: Embedded Software

Lecture 2:
C for Embedded Systems I

C for embedded systems

- The course covers C language constructs frequently used in embedded software development
  - Preprocessor directives
    - Informs the compiler about the hardware
  - Mixing C and assembly
    - Often a necessity in embedded systems
  - Pointers
    - Used to access memory and input and output devices
  - Bit manipulation
    - Used to handle hardware-level details, input and output
  - Structures, unions
    - In the context of pointers and bit manipulation
- C or C++ knowledge is a prerequisite for this course

Lecture outline

- Why C?
- Preprocessor directives
- Mixing C and assembly
- Pointers
  - Pointer arithmetic
- Structures
- Unions

Lecture outline

- **Why C?**
- Preprocessor directives
- Mixing C and assembly
- Pointers
  - Pointer arithmetic
- Structures
- Unions

History of C

- Evolved by Ritchie from two previous programming languages, BCPL and B
- Used to develop UNIX
- Used to write modern operating systems
- Hardware independent (portable)

C standardization

- Many slight variations of C existed, and were incompatible
- Committee formed to create a "unambiguous, machine-independent" definition
- Standard created in 1989, updated in 1999

Why use C for writing embedded software?

- Small, simple to learn
- Available for almost all processors used today

- C: very "low-level" high-level language
  - Gives embedded programmers an extraordinary degree of direct hardware control without sacrificing the benefits of high-level languages

Why use C for writing embedded software?

- Kernighan and Ritchie, *The C Programming Language*:
  - C is a relatively "low level" language. This characterization is not pejorative; it simply means that C deals with the same sort of objects that most computers do. These may be combined and moved about with the arithmetic and logical operators implemented by real machines.
  - Because the data types and control structures provided by C are supported directly by most computers, the run-time library required to implement self-contained programs is tiny. The standard library functions are only called explicitly, so they can be avoided if they are not needed.

Lecture outline

- Why C?
- **Preprocessor directives**
- Mixing C and assembly
- Pointers
  - Pointer arithmetic
- Structures
- Unions

Preprocessing directives

- Preprocessing
  - Occurs before a program is compiled
  - Inclusion of other files
  - Definition of symbolic constants and macros
  - Conditional compilation of program code
  - Conditional execution of preprocessor directives
- Format of preprocessor directives
  - Lines begin with #

The #include preprocessor directive

- #include
  - Copy of a specified file included in place of the directive
  - #include <filename>
    - Searches standard library for file
    - Use for standard library files
  - #include "filename"
    - Searches current directory, then standard library
    - Use for user-defined files
  - Used for:
    - Programs with multiple source files to be compiled together
    - Header file – has common declarations and definitions (classes, structures, function prototypes)
      - #include statement in each file

The #define preprocessor directive: Symbolic constants

- #define
  - Preprocessor directive used to create symbolic constants and macros
  - Symbolic constants
    - When program compiled, all occurrences of symbolic constant replaced with replacement text
  - Format

    #define *identifier replacement-text*
  - Example:

    #define PI 3.14159
  - Everything to right of identifier replaces text

    #define PI = 3.14159
    - Replaces “PI” with "= 3.14159”

The #define preprocessor directive: Macros

- Macro
  - Operation defined in #define
  - A macro with arguments has its arguments substituted for replacement text, when the macro is expanded
  - Performs a text substitution – no data type checking
  - The macro

```
#define CIRCLE_AREA( x ) ( PI * ( x ) * ( x ) )
```

would cause

```
area = CIRCLE_AREA( 4 );
```

to become

```
area = ( 3.14159 * ( 4 ) * ( 4 ) );
```

Examples: Pitfalls

It is left as an exercise to find out what (may) become wrong with the definition below:

```
#define POW(x) x*x
#define CIRCLE_AREA( x )  PI *  x  * x
#define RECTANGLE_AREA( x, y )   x  *  y
```

Reference:

http://gcc.gnu.org/onlinedocs/cpp/Macros.html

Conditional compilation

- Control preprocessor directives and compilation
- Cast expressions, sizeof, enumeration constants cannot be evaluated in preprocessor directives
- Structure similar to if

```
#if !defined( NULL )
  #define NULL 0
#endif
```

- Determines if symbolic constant NULL has been defined
  - If NULL is defined, defined(NULL) evaluates to 1
  - If NULL is not defined, this function defines NULL to be 0
- Every #if must end with #endif
  - #ifdef short for #if defined(name)
  - #ifndef short for #if !defined(name)

Conditional compilation, cont.

- Use for commenting out code
- C does not allow nested comments

```
/* First layer
    /* Second layer */
*/
```

- Use you can use the #if .. #endif combination to cause the preprocessor to avoid compiling any portion of your code by using a condition that will never be true.

```
#if 0
        code commented out
#endif
```

  - To enable code, change 0 to 1

Conditional compilation, cont.

- Other statements
  - #elif – equivalent of else if in an if statement
  - #else – equivalent of else in an if statement

Conditional compilation, cont.

- Debugging

```
#define DEBUG  1
#ifdef DEBUG
    cerr << "Variable x = " << x << endl;
#endif
```

  - Defining DEBUG to 1 enables code
  - After code corrected, remove #define statement
  - Debugging statements are now ignored

The #error Preprocessor Directives

- #error tokens
  - Tokens are sequences of characters separated by spaces
    - "I like C" has 3 tokens
  - Displays a message including the specified tokens as an error message
  - Stops preprocessing and prevents program compilation
- The directive ‘#error’ causes the preprocessor to report a fatal error. The tokens forming the rest of the line following ‘#error’ are used as the error message.
  - E.g.,

```
#if !defined(FOO) && defined(BAR)
  #error "BAR requires FOO."
  #endif
```

The # and ## operators

- #
  - Causes a replacement text token to be converted to a string surrounded by quotes
  - The statement

    ```
    #define HELLO( x ) printf( “Hello, ” #x  “\n” );
    ```

  would cause

    ```
    HELLO( John )
    ```

  to become

    ```
    printf( “Hello, ” “John” “\n” );
    ```

  - Strings separated by whitespace are concatenated when using printf

The # and ## operators, cont.

- ##
  - Concatenates two tokens
  - The statement

    ```
    #define TOKENCONCAT( x, y )  x ## y
    ```

  would cause

    ```
    TOKENCONCAT( O, K )
    ```

  to become

    ```
    OK
    ```

Line numbers

- #line
  - Renumbers subsequent code lines, starting with integer value
  - File name can be included
  - #line 100 "myFile.c"
    - Lines are numbered from 100 beginning with next source code file
    - Compiler messages will think that the error occurred in "myfile.C"
    - Makes errors more meaningful
    - Line numbers do not appear in source file because of this!

Predefined symbolic constants

- Four predefined symbolic constants
  - Cannot be used in #define or #undef

Symbolic constant	Description
__LINE__	The line number of the current source code line (an integer constant).
__FILE__	The presumed name of the source file (a string).
__DATE__	The date the source file is compiled (a string of the form **"Mmm dd yyyy"** such as **"Jan 19 2001"**).
__TIME__	The time the source file is compiled (a string literal of

Examples: Macro definitions

```
#define SIZE 128
#define POW(x) ((x)*(x))
#define DEBUG(format, ...) \
  printf(format, ## __VA_ARGS__)
#define DUMP(int_var) \
  printf("%s = %d\n", #int_var, int_var)
#define WHERE \
  printf("'%s' at %d\n", __FILE__, __LINE__)
#define DEREF_AND_INC(ptr) do { \
    if ( (ptr) != NULL )        \
      *(ptr) += 1;              \
  } while (0)
```

Examples: Macro use

```
int main()
{
  int array[SIZE];
  POW(array[0] + array[1]);
  /* Disable all DEBUG by changing macro */
  DEBUG(”%s\n”, ”I reached the top.”);
  /* Easy to get nice print of variable. */
  DUMP(array[4]);
  WHERE; /* Prints file and line. */
  /* Note that adding the ; is valid !! */
  DEREF_AND_INC(array + 10);
}
```

- Use of macros in embedded systems – a specific example
  - An example in p100 of the textbook: Programming Embedded Systems by Barr and Massa

Lecture outline

- Why C?
- Preprocessor directives
- **Mixing C and assembly**
- Pointers
  - Pointer arithmetic
- Structures
- Unions

Inline assembly

- Compiler dependent
  - DJGPP (C development environment for Intel 80386)
    - __asm__ __volatile__ { /* assembly code */ }
      - __asm__ instructs the compiler to treat the parameters of the statement as pure assembly and to pass them to the assembler as written. ;
      - __volatile__ is an optional statement which instructs the compiler not to move opcodes around
  - Microsoft C
    - _asm mov ah, 2
    - _asm {
      /* assembly code */
      }

x86 assembler

- Intel syntax
  - Used by NASM assembler
  - Opcode *Destination* Source (order as in ”Y = X” in C)
  - Hexadecimal constants end with ”h” as in 1234h
  - Operand prefix determine size (byte ptr, word ptr, dword ptr)
  - Memory adressing like section:[base + index*scale + disp]

C – Assembly interfacing

- EBP is base pointer (helps us to point to things in stack)
- ESP is current stack pointer
- PUSH send contents to top of stack
- POP retrieve contents from top of stack

Write C function in assembly

```
print(const char* str, int size);

print:
  PUSH EBP       ; save previous stack frame
  MOV EBP, ESP   ; save current stack frame

  MOV ECX,[EBP+8]  ; read parameter 'str'
  MOV EDX,[EBP+12] ; read parameter 'size'
  ; do function stuff here

  MOV ESP, EBP   ; restore stack frame
  POP EBP        ; restore previous frame
  RET
```

Calling convention

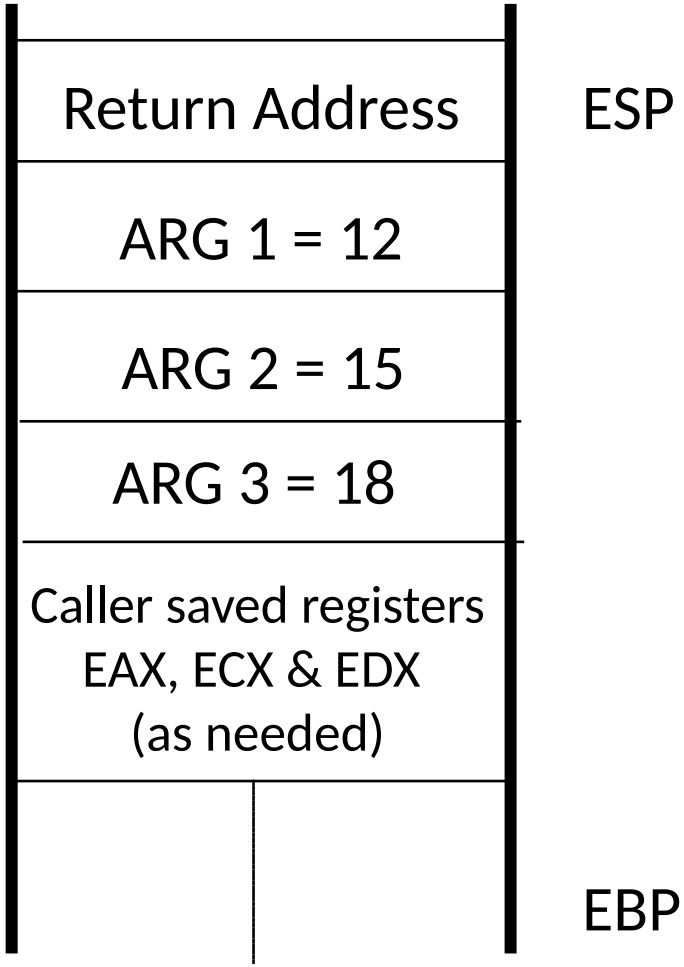
- We want the computer to serve different procedures one after another
- When a caller calls a procedure, the program must follow some steps
  - Put parameters in a place in memory so that the callee (the called procedure) may access them
  - Transfer control to the callee
  - Acquire space on memory to store local variables, if needed
  - Do computation
  - Return control to point of origin in caller
- Calling convention is the set of rules that compilers and programmers must follow to achieve the above

Caller (before it calls)

- Example,
  - Caller is the main function
  - It will call a function called foo
  - a = foo(12, 15, 18)

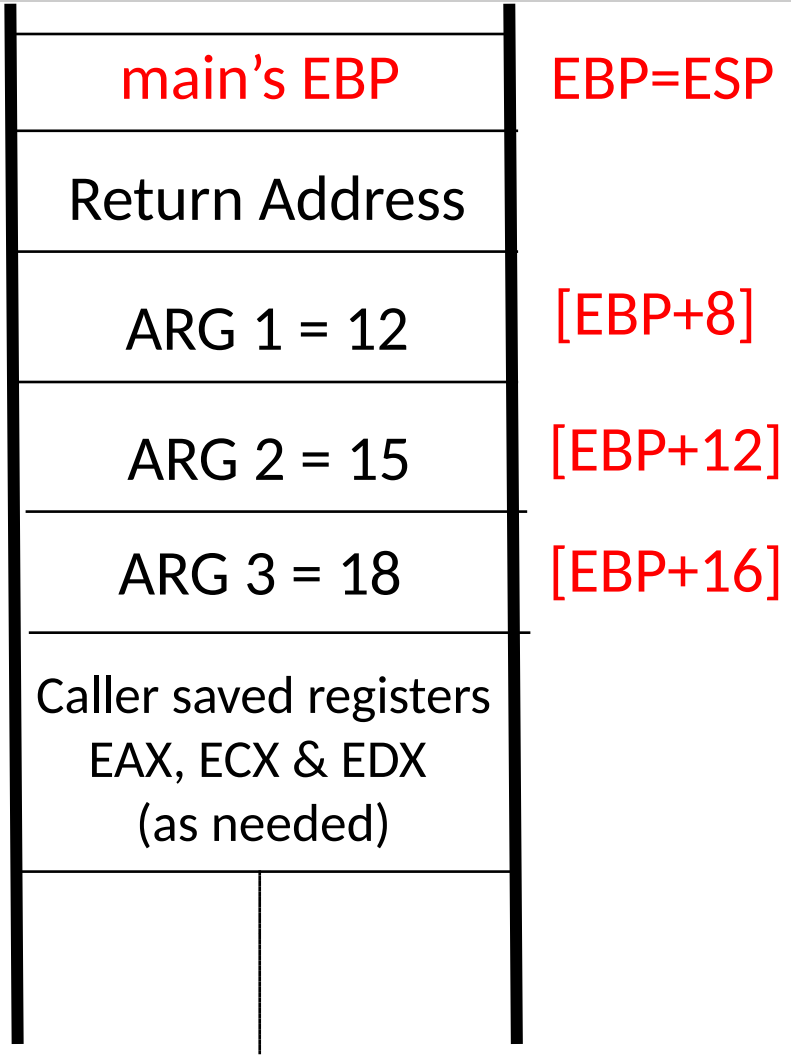
Caller (before it calls)

- Main (caller) is using ESP and EBP
- First, main pushes, EAX, ECX and EDX (only if they need to be preserved)
- Next, it pushes the arguments (last argument first) i.e.,
  - push dword 18
  - Push dword 15
  - Push dword 12
- Finally, main can execute
  - call foo
  - Then, the return address (contents of register EIP, i.e., the program counter) is pushed to the top of the stack



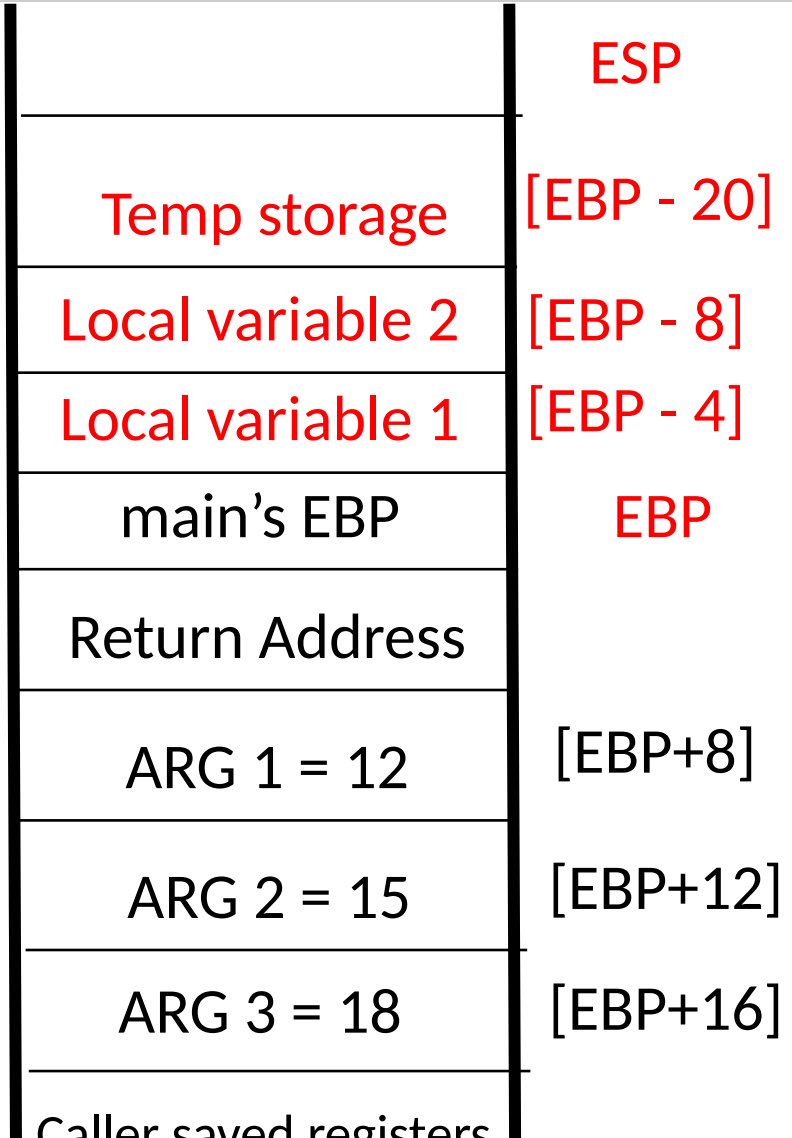
Callee (after it was called)

- First, foo (callee) must setup its own stack frame. The EBP register was pointing to somewhere in main's stack frame. This must be preserved. So, we push it.
- Then, the contents of ESP are transferred to EBP. ESP is freed to do other things and EBP is now the base pointer for foo. So, we can point to things in foo's stack with an offset from EBP
- The above two steps are:
  - Push EBP
  - Mov EBP, ESP
- 4 bytes for main's EBP and 4 bytes for return address. That's why 8 bytes offset to first argument.



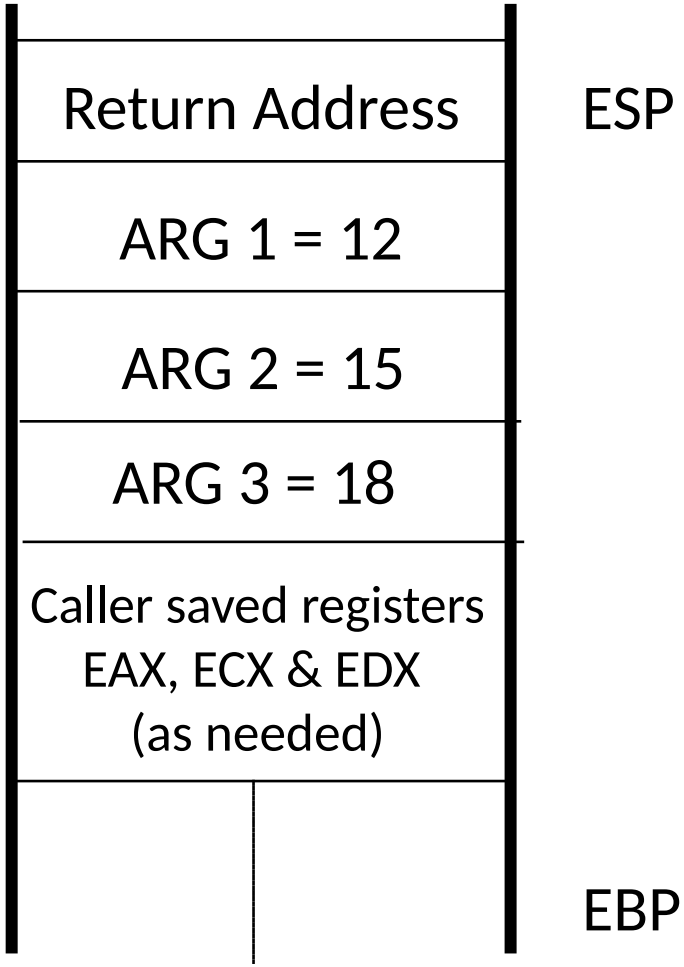
Callee (after it was called, contd)

- Foo must allocate space for temporary variables that cannot be stored in registers as well as other storage memory
- If, e.g, foo has 2 variables of type int (4 bytes), plus it needs 12bytes. Total = 20 bytes.
- Allocate 20 bytes in the stack, by adjusting ESP
  - sub esp, 20
- Finally foo must preserve EBX, ESI and EDI if they are being used
- Now, foo can be executed. ESP can go up and down but the EBP will remain same.
- Maybe other functions are called but always EBP is restored.



Callee (before it returns)

- Store the computed (return) value in EAX
- Restore values of EBX, ESI and EDI, if needed.
- Now, release area used for local storage and temporary registers spillover. Then, pop the return address of main to EBP
  - Mov ESP, EBP
  - Pop EBP
- Finally, just return
  - Ret



Caller (after returning)

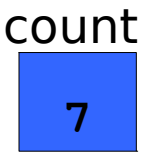
- Control returns to caller (main)
- The arguments passed to foo are not needed any more, so adjust ESP
  - Add ESP , 12
- Save the EAX (returned value) in appropriate memory location
- Main pops, EAX, ECX and EDX (only if they were preserved before the call)

Lecture outline

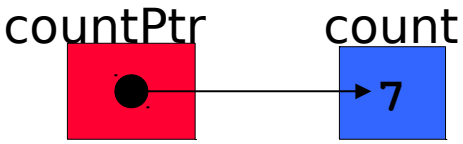
- Why C?
- Preprocessor directives
- Mixing C and assembly
- **Pointers**
  - Pointer arithmetic
- Structures
- Unions

Pointer variable definitions and initialization

- Pointer variables
  - Contain memory addresses as their values
  - Normal variables contain a specific value (direct reference)



  - Pointers contain address of a variable that has a specific value (indirect reference)
  - Indirection – referencing a pointer value



Pointer variable definitions and initialization, cont.

- Pointer definitions
  - ***** used with pointer variables

    ```
    int *myPtr;
    ```

  - Defines a pointer to an int (pointer of type int *)
  - Multiple pointers require using a * before each variable definition

    ```
    int *myPtr1, *myPtr2;
    ```

  - Can define pointers to any data type
  - Initialize pointers to 0, NULL, or an address
    - 0 or NULL – points to nothing (NULL preferred)

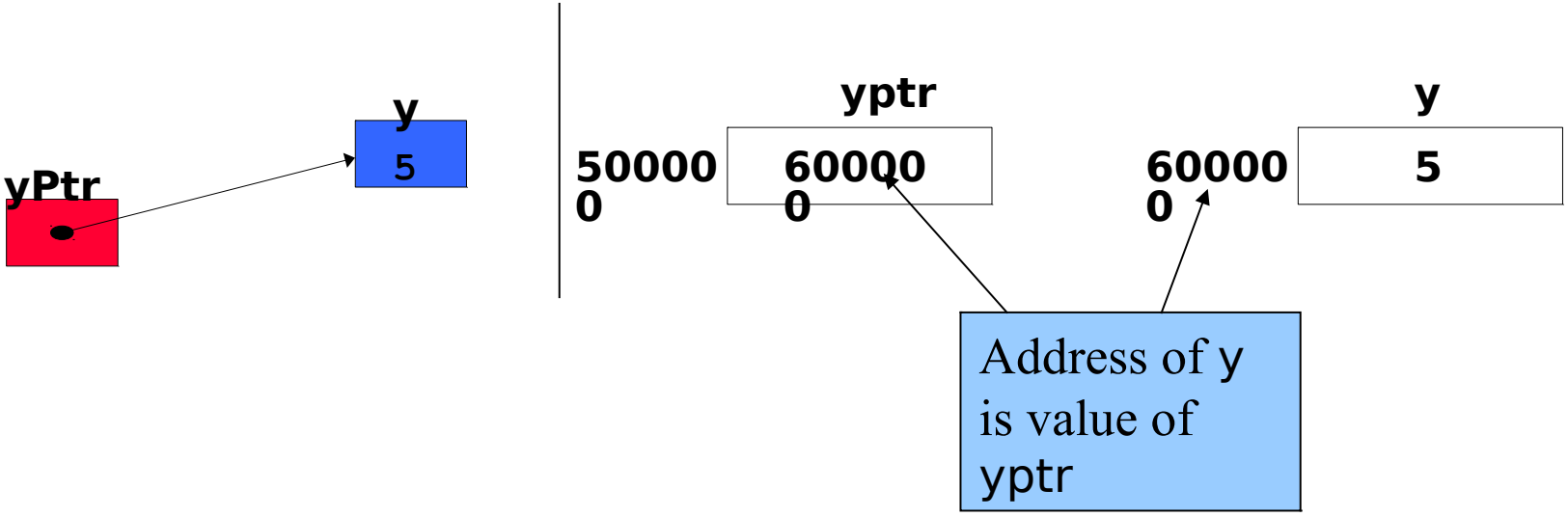
Display example

- Alphanumeric color display
  - Memory mapped at address 0xB800
  - 80 characters on each line
  - Each character consists of two bytes
    - First byte: ASCII code
    - Second byte: foreground+background colors
  - char *p = (char *)(0xB8000+2*(80*row+col));
    p = value; / e.g., display ‘A’ on screen */

Pointer operators

- & (address operator)
  - Returns address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;     /* yPtr gets address of y */
yPtr “points to” y
```



Pointer operators, cont.

- * (indirection/dereferencing operator)
  - Returns a synonym/alias of what its operand points to
  - *yptr returns y (because yptr points to y)
  - * can be used for assignment
    - Returns alias to an object

```
*yptr = 7;  /* changes y to 7 */
```

  - Dereferenced pointer (operand of *) must be an lvalue (no constants)
- * and & are inverses
  - They cancel each other out

Pointer operators, cont.

Operators								Associativity	Type
()	[]							left to right	highest
+	-	++	--	!	*	&	(type)	right to left	unary
*	/	%						left to right	multiplicative
+	-							left to right	additive
<	<=	>	>=					left to right	relational
==	!=							left to right	equality
&&								left to right	logical and
||								left to right	logical or
?:								right to left	conditional

Calling functions by reference

- Call by reference with pointer arguments
  - Pass address of argument using & operator
  - Allows you to change actual location in memory
  - Arrays are not passed with & because the array name is already a pointer
- * operator
  - Used as alias/nickname for variable inside of function

```
void double( int *number )
{
  *number = 2 * ( *number );
}
```

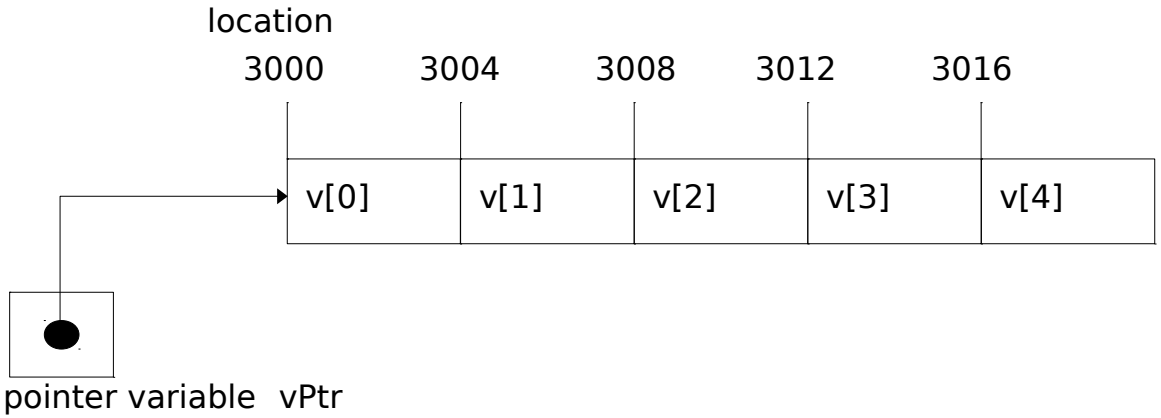
  - *number used as nickname for the variable passed

Pointer expressions and pointer arithmetic

- Arithmetic operations can be performed on pointers
  - Increment/decrement pointer (++ or --)
  - Add an integer to a pointer(+ or += , - or -=)
  - Pointers may be subtracted from each other
  - Operations meaningless unless performed on an array

Pointer expressions and pointer arithmetic, cont.

- 5 element int array on machine with 4 byte ints
  - vPtr points to first element v[0]
    - at location 3000 (vPtr = 3000)
  - vPtr += 2; sets vPtr to 3008
    - vPtr points to v[2] (incremented by 2), but the machine has 4 byte ints, so it points to address 3008



Pointer expressions and pointer arithmetic, cont.

- Subtracting pointers
  - Returns number of elements from one to the other. If
    ```
    vPtr2 = v[ 2 ];
    vPtr = v[ 0 ];
    ```
  - vPtr2 - vPtr would produce 2
- Pointer comparison (<, == , >)
  - See which pointer points to the higher numbered array element
  - Also, see if a pointer points to 0

Pointer expressions and pointer arithmetic, cont.

- Pointers of the same type can be assigned to each other
  - If not the same type, a cast operator must be used
  - Exception: pointer to void (type void *)
    - Generic pointer, represents any type
    - No casting needed to convert a pointer to void pointer
    - void pointers cannot be dereferenced

Trivia

- A survey from 2005 to 2012
  - ??% develop embedded software in C
  - ??% in C++
  - ??% in Java
  - ??% programmers used Assembly programming in their embedded programming
- Source : Embedded.com

Trivia

- A survey from 2005 to 2012
  - 65% develop embedded software in C
  - 20% in C++
  - Less than 5% in Java
  - Over 60% programmers used Assembly programming in their embedded programming

The relationship between pointers and arrays

- Arrays and pointers closely related
  - Array name like a constant pointer
  - Pointers can do array subscripting operations
- Define an array b[5] and a pointer bPtr
  - To set them equal to one another use:

    bPtr = b;

    - The array name (b) is actually the address of first element of the array b[5]

      bPtr = &b[0]

    - Explicitly assigns bPtr to address of first element of b

The relationship between pointers and arrays, cont.

- Element b[3]
  - Can be accessed by *(bPtr + 3)
    - Where n is the offset. Called pointer/offset notation
  - Can be accessed by bptr[3]
    - Called pointer/subscript notation
    - bPtr[3] same as b[3]
  - Can be accessed by performing pointer arithmetic on the array itself

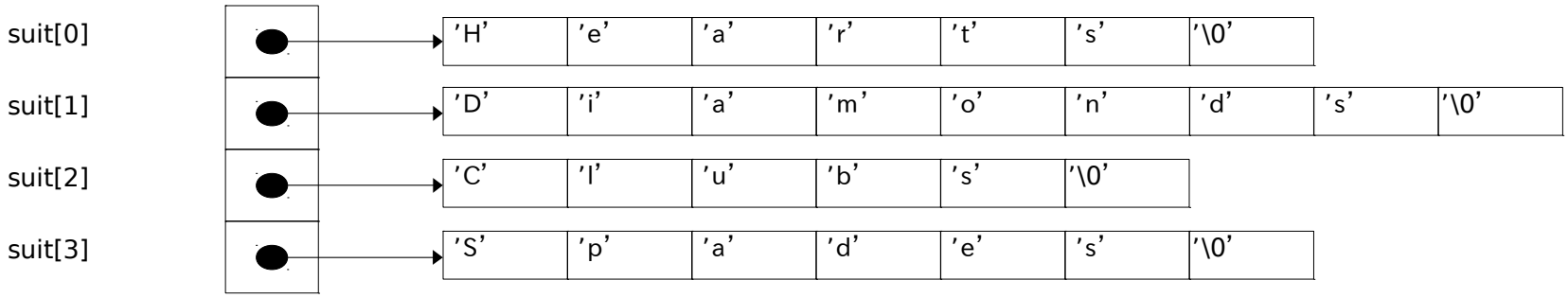
    *(b + 3)

Arrays of pointers

- Arrays can contain pointers
- For example: an array of strings

```
char *suit[ 4 ] = { "Hearts", "Diamonds",
   "Clubs", "Spades" };
```

  - Strings are pointers to the first character
  - char * – each element of suit is a pointer to a char
  - The strings are not actually stored in the array suit, only pointers to the strings are stored



  - suit array has a fixed size, but strings can be of any size

Display example, cont.

- char *color_display_buffer = (char *)0xB8000;
  color_display_buffer[2*(80*row+col)] = ‘A’;

  - color_buffer_display[i] is a reference to the i^{th} row of cells
  - color_buffer_display[i][j] selects the j^{th} cell of the i^{th} row

  - Each character is two bytes:
    - color_display_buffer[row][col][0] = ‘A’;

Pointers to functions

- Pointer to function
  - Contains address of function
  - Similar to how array name is address of first element
  - Function name is starting address of code that defines function
- Function pointers can be
  - Passed to functions
  - Stored in arrays
  - Assigned to other function pointers

Pointers to functions, cont.

- Example: bubblesort
  - Function bubble takes a function pointer
    - bubble calls this helper function
    - this determines ascending or descending sorting
  - The argument in bubblesort for the function pointer:

    ```
    int ( *compare )( int a, int b )
    ```

    tells bubblesort to expect a pointer to a function that takes two ints and returns an int
  - If the parentheses were left out:

    ```
    int *compare( int a, int b )
    ```

    - Defines a function that receives two integers and returns a pointer to a int

Lecture outline

- Why C?
- Preprocessor directives
- Mixing C and assembly
- Pointers
  - Pointer arithmetic
- **Structures**
- Unions

Structures

- Structures
  - Collections of related variables (aggregates) under one name
    - Can contain variables of different data types
  - Commonly used to define records to be stored in files
  - Combined with pointers, can create linked lists, stacks, queues, and trees
  - Can hold the data associated to a hardware device

Structure definitions

- Example

```
struct card {
 char *face;
 char *suit;
};
```

  - struct introduces the definition for structure card
  - card is the structure name and is used to declare variables of the structure type
  - card contains two members of type char *
    - These members are face and suit

Structure definitions, cont.

- struct information
  - A struct cannot contain an instance of itself
  - Can contain a member that is a pointer to the same structure type
  - A structure definition does not reserve space in memory
    - Instead creates a new data type used to define structure variables

- Definitions
  - Defined like other variables:

```
card oneCard, deck[ 52 ], *cPtr;
```

  - Can use a comma separated list:

```
struct card {
   char *face;
   char *suit;
} oneCard, deck[ 52 ], *cPtr;
```

Structure definitions, cont

- Valid operations
  - Assigning a structure to a structure of the same type
  - Taking the address (&) of a structure
  - Accessing the members of a structure
  - Using the sizeof operator to determine the size of a structure

Initializing structures

- Initializer lists
  - Example:

    ```
    card oneCard = { "Three", "Hearts" };
    ```
- Assignment statements
  - Example:

    ```
    card threeHearts = oneCard;
    ```
  - Could also define and initialize threeHearts as follows:

    ```
    card threeHearts;
    threeHearts.face = “Three”;
    threeHearts.suit = “Hearts”;
    ```

Accessing members of structures

- Accessing structure members
  - Dot operator (**.**) used with structure variables

```
card myCard;
printf( "%s", myCard.suit );
```

  - Arrow operator (**->**) used with pointers to structure variables

```
card *myCardPtr = &myCard;
printf( "%s", myCardPtr->suit );
```

  - myCardPtr->suit is equivalent to

```
( *myCardPtr ).suit
```

Using structures with functions

- Passing structures to functions
  - Pass entire structure
    - Or, pass individual members
  - Both pass call by value
- To pass structures call-by-reference
  - Pass its address
  - Pass reference to it
- To pass arrays call-by-value
  - Create a structure with the array as a member
  - Pass the structure

typedef

- typedef
  - Creates synonyms (aliases) for previously defined data types
  - Use typedef to create shorter type names
  - Example:

```
typedef struct Card *CardPtr;
```

  - Defines a new type name CardPtr as a synonym for type struct Card *
  - typedef does not create a new data type
    - Only creates an alias

typedefs

```
unsigned long int count ;
```

versus

```
typedef unsigned long int DWORD32 ;
DWORD32 count ;
```

typedefs and #defines

```
typedef unsigned char          BYTE8 ;
typedef unsigned short int     WORD16 ;
typedef unsigned long int      DWORD32 ;


typedef int                    BOOL ;
#define FALSE                  0
#define TRUE                   1
```

Lecture outline

- Why C?
- Preprocessor directives
- Mixing C and assembly
- Pointers
  - Pointer arithmetic
- Structures
- **Unions**

Unions

- union
  - Memory that contains a variety of objects over time
  - Only contains one data member at a time
  - Members of a union share space
  - Conserves storage
  - Only the last data member defined can be accessed
- union definitions
  - Same as struct

```
union Number {
  int x;
  float y;
};
union Number value;
```

```
union Data
{
   int i;
   float f;
   char str[20];
} data;
```

The memory occupied by a union will be large enough to hold the largest member of the union.

For example, in above example Data type will occupy 20 bytes.

Unions

- Valid union operations
  - Assignment to union of same type: =
  - Taking address: &
  - Accessing union members: .
  - Accessing members using pointers: ->

Variant access with pointers, casts, & subscripting

- Given an address, we can cast it as a pointer to data of the desired type, then deference the pointer by subscripting.
- Without knowing the data type used in its declaration, we can read or write various parts of an object named *operand* using:

```
((BYTE8 *) &operand)[k]
```

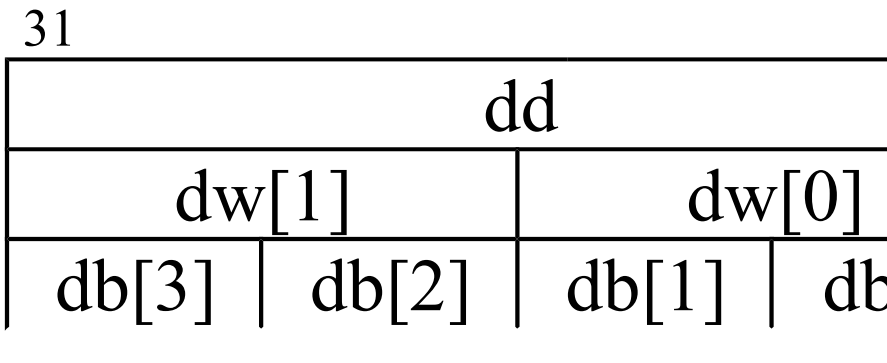
Variant access with pointers, casts, & subscripting, cont.

```
typedef struct KYBD_INFO
        {
        BYTE8     lo ;
        BYTE8     hi ;
        WORD16  both ;
        } KYBD_INFO ;
```

```
BOOL Kybd_Flags_Changed(KYBD_INFO *kybd)
        {
        …….
        kybd->both   = ((WORD16  *) &new_flags)[0] ;
        kybd->lo     = ((BYTE8  *)    & new_flags)[0] ;
        kybd->hi     = ((BYTE8  *)     &new_flags)[1] ;

        if (kybd->both == old_flags) return FALSE ;
        old_flags = kybd->both ;

        return TRUE ;
        }
```

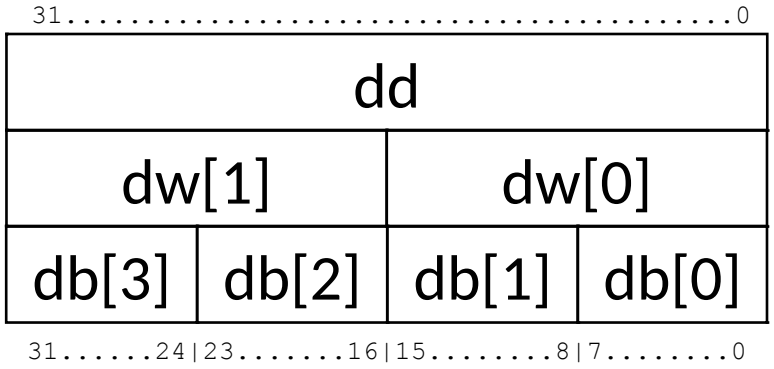
Variant access with unions

```
union {
   unsigned long   dd ;
   unsigned short  dw[2] ;
   unsigned char   db[4] ;
```



Endianness

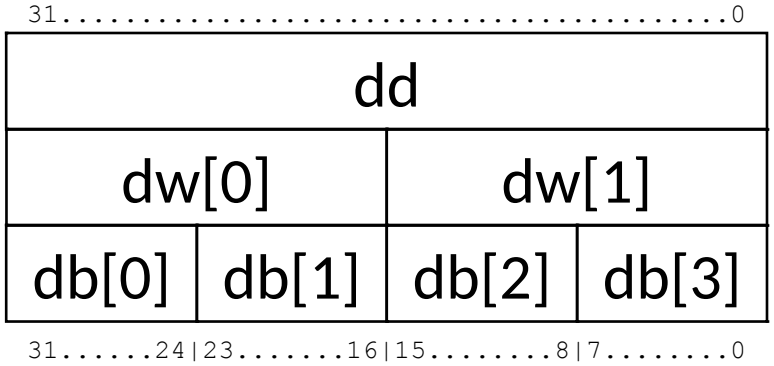
```
union
{
    unsigned long dd;
    unsigned short dw[2];
    unsigned char db[4];
};
```



Little endian

vs

Big endian



Endianness differ depending on architecture.
X86: little
Motorola, sparc: big

Endianness

- **Big-endian** systems are systems in which the *most significant byte* of the word is stored in the *smallest address* given and the least significant byte is stored in the largest. In contrast, **little endian** systems are those in which the *least significant byte* is stored in the *smallest address*.

Why is Endianness important for embedded software developers?

- Think about communication between two machine that have different Endianness
- One machine writes integers to a file and another machine with opposite Endianness reads it.
- Sending numbers over network between two machines with different Endianess. Think about serial communication when we split the data into mulitple chunks !!

Trivia!

- Where does this term ‘Endian’ come from?

Trivia!

- Excellent read: http://www.ietf.org/rfc/ien/ien137.txt
- Quote:
- “It may be interesting to notice that the point which Jonathan Swift tried to convey in Gulliver's Travels in exactly the opposite of the point of this note.
- Swift's point is that the difference between breaking the egg at the little-end and breaking it at the big-end is trivial. Therefore, he suggests, that everyone does it in his own preferred way.
- We agree that the difference between sending eggs with the little- or the big-end first is trivial, but we insist that everyone must do it in the same way, to avoid anarchy. Since the difference is trivial we may choose either way, but a decision must be made.”

Variant access with unions, cont.

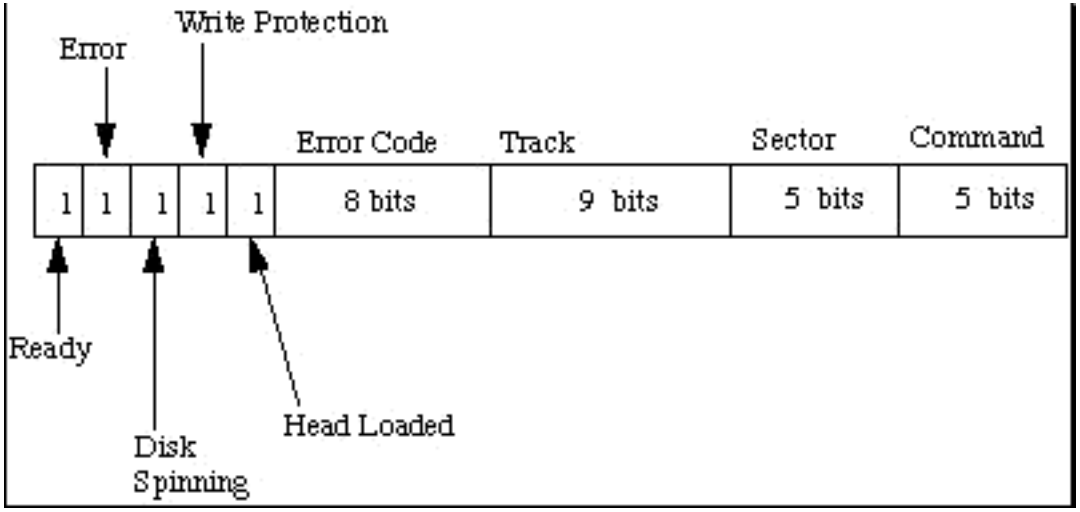
```
typedef union VARIANT {
        BYTE8           b[2] ;
        WORD16          w ;
} VARIANT ;

BOOL Kybd_Flags_Changed(KYBD_INFO *kybd)
        {
        static WORD16 old_flags = 0xFFFF ;
        VARIANT *flags = (VARIANT *) malloc(sizeof(VARIANT)) ;

        dosmemget(0x417, sizeof(VARIANT), (void *) flags) ;

        status->both        = flags->w ;
        status->lo          = flags->b[0] ;
        status->hi          = flags->b[1] ;
        free(flags) ;

        if (status->both == old_flags) return FALSE ;
        old_flags = status->both ;
        return TRUE ;
        }
```

Bitfield

- In embedded systems, storage is at a premium
- It may be necessary to pack several objects into one word
- Bitfields allow single bit objects
- They must be part of structure

Bitfield example

- Embedded systems must communicate with peripherals at low-level.
- A register of a disk controller, for example, has several fields.



- How can we represent this is memory compactly?

Bitfield example

```
struct DISK_REGISTER {
    unsigned int ready:1;
    unsigned int error_occured:1;
    unsigned int disk_spinning:1;
    unsigned int write_protect:1;
    unsigned int head_loaded:1;
    unsigned int error_code:8;
    unsigned int track:9;
    unsigned int sector:5;
    unsigned int command:5;
    };
```

- Bit fields must be part of a structure/union – stipulated by the C standard