CpE 213
Digital Systems Design
C Programming Review
Structured Programming

Lecture 17

Friday 10/21/2005

Overview

- Project 1
- C Programming review
- Structured programming
- Note: We will be skipping some example slides today.

Please read these lecture notes in their entirety.

C Programming Review

Some slides adapted from C reviews by Dr. Warter-Perez and
Mr. Anubhav Gupta

Reference Links for C

- An online C Primer:
  http://occs.cs.oberlin.edu/faculty/jdonalds/341/CPrimer.html
- Another C Primer:
  - http://www.vectorsite.net/tscpp.html
- A more detailed reference:
  - http://www.cs.cf.ac.uk/Dave/C/CE.html
- A number of books on the topic are listed in your course syllabus.
- See me for a handout on basics of C.

C Basics - Variables

- Variables have a data type and name or identifier.
- Identifiers
  - Have the following restrictions:
    - Must start with a letter or underscore (_)
    - Must consist of only letters, numbers or underscore
    - Must not be a keyword
  - Have the following conventions:
    - All uppercase letters are used for constants
    - Variable names are meaningful – thus, often multi-word
      - Convention 1: alignment_sequence
      - Convention 2: AlignmentSequence

C Basics – Data Types (1)

- 3 basic data types: integer, float, char
  - Integer (int) – represent whole numbers
    - long (32-bits same as default), short (16-bits)
      - System dependent
    - signed (positive and negative, default), unsigned (positive)
  - Ex 1: define an integer variable y
    - `int y;     // initialized to garbage`
  - Ex 2: define an unsigned short integer variable month initialized to 4 (April)
    - `unsigned short int month = 4;`

C Basics – Data Types (2)

- Floating point – represent real numbers
  - IEEE Standards
    - Single-precision (float, 32-bits)
    - Double-precision (double, 64-bits)
  - Ex 1: define a single-precision floating-point variable named error_rate and initialize to 3.5
    - `float error_rate = 3.5;`
  - Ex 2: define a double-precision floating-point variable named score and initialize it to .004 using scientific notation
    - `double score = 4e-3;`

C Basics – Data Types (3)

- Character – represent text
  - ASCII – American Standard Code for Information Interchange
  - Represents characters, numbers, punctuation, spacing and special non-printable control characters
  - Example ASCII codes: 'A' = 65, 'B' = 66, … 'a' = 97, 'b' = 98, '\n' = 10
  - Ex 1: define a character named AminoAcid and initialize it to 'C'
    - `char AminoAcid = 'C';`
    - `char AminoAcid = 67;      // equivalent`

Summary of Data Types

data type	size (bytes)	values (range)
char	1	-128 to 127
short	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,483,647
float	4	3.4E+/-38 (7 digits)
double	8	1.7E+/-308 (15 digits long)

Warning: Using the float and double data types is <u>very</u> difficult on the 8051.

C symbol definition

- Normally put into include files (*.h)
- Some examples (see reg51.h):
  - sfr P0   = 0x80;
  - sfr P1   = 0x90;
  - sfr P2   = 0xA0;
  - sfr PSW = 0xD0;
  - sbit CY   = 0xD7;
  - sbit P0_1 = P0^1; /* same as 81h */

Bit operator

Arithmetic Operators

Operator		Example
		`int x, y=5, z=3;`
+	add	`x = y + z;` x = 8
-	subtract	`x = y – z;` x = 2
*	multiply	`x = y * z;` x = 15
/	divide	`x = y / z;` x = 1
%	modulus	`x = y % z;` x = 2

Auto Increment and Decrement

x = 3

- Pre-increment/decrement
  - `y = ++ x;` *equivalent to* `x = x+1;` x = 4
    `y = x;` y = 4
  - `y = --x;` *equivalent to* `x = x–1;` x = 2
    `y = x;` y = 2
- Post-increment/decrement
  - `y = x++;` *equivalent to* `y = x;` y = 3
    `x = x+1;` x = 4
  - `y = x--;` *equivalent to* `y = x;` y = 3
    `x = x–1;` x = 2

Relational and Logical Operators

- Relational operators
  - == equal
  - != not equal
  - > greater than
  - >= greater than or
  - equal
  - < less than
  - <= less than or equal
- Logical operators
  - && and
  - || or
  - ! not

Relational Operators

- Assume x is 1, y is 4, z = 14

Expression	*Value*	*Interpretation*
`x < y + z`	1	True
`y == 2 * x + 3`	0	False
`z <= x + y`	0	False
`z > x`	1	True
`x != y`	1	True

Logical Operators

- Assume x is 1, y is 4, z = 14

Expression	*Value*	*Interpretation*
`x<=1 && y==3`	0	False
`x<= 1 || y==3`	1	True
`!(x > 1)`	1	True
`!x > 1`	0	False
`!(x<=1 || y==3)`	0	False

Control Flow Summary

- if-else: decision making
- else-if: multi-way branch
- switch: another multi-way branch
- while and for: test at top of loop
- do while: test at bottom of loop
- break and continue
- goto and labels (avoid!)

if Statement

- if (*expression*)

  action

Example:

```
char a1 = 'A', a2
  = 'C';
int match = 0;
if (a1 == a2) {
  match++;
}
```

if-else Statement

- if (*expression*)

  action 1

  else

  action 2

Example:

```
char a1 = 'A', a2 = 'C';
int match = 0, gap = 0;
if (a1 == a2) {
   match++;
} else {
   gap++;
}
```

Note: Also see the “switch” statement.

for Statement

for(*expr1; expr2; expr3*)
action

- *Expr1 – defines initial conditions*
- *Expr2 – tests for continued looping*
- *Expr3 – updates loop*

Example

```
sum = 0;
for(i = 1; i <= 4; i++)
   sum = sum + 1;
```

Iteration 1: sum=0+1=1

Iteration 2: sum=1+2=3

Iteration 3: sum=3+3=6

Iteration 4: sum=6+4=10

while Statement

```
while (expression)
   action
```

<u>Note: Read "do while" on your own.</u>

<u>Example</u>

```
int x = 0;
while(x != 3) {
   x = x + 1;
}
```

/ 2

Infinite loop!

Iteration 1: x=0+1=1

Iteration 2: x=1+1=2

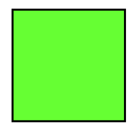
Iteration 3: x=2+1=3

Iteration 4: don't exec

1-D Arrays

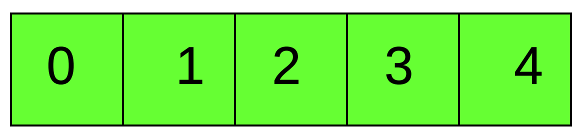
- char amino_acid;
  - Defines one amino_acid as a character

1 cell



- char sequence[5];
  - Defines a sequence of 5 elements of type character (where each element may represent an amino acid)

5 cells with indices



Initializing Arrays

- char seq [5] = “ACTG”;

5 cells with values

'A'	'C'	'T'	'G'	'\0'

seq[0] = 'A'

seq[1] = 'C'

...

- float hydro[6] = {-0.2, 0, -0.67, -3.5, 2.8};

5 cells with values

-.2	0	-.67	-3.5	2.8	0

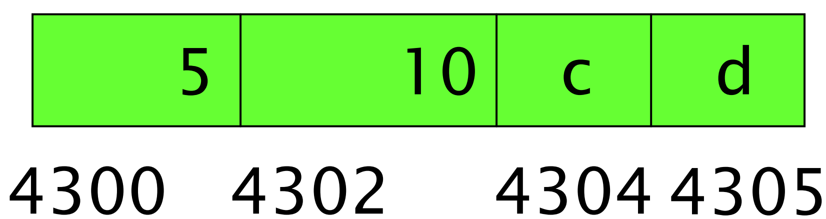
hydro['A' - 'A'] = -.2   hydro['C' - 'A'] = -.67   hydro[5] = 0

- No initialization – each cell has “garbage” – unknown value

Pointers

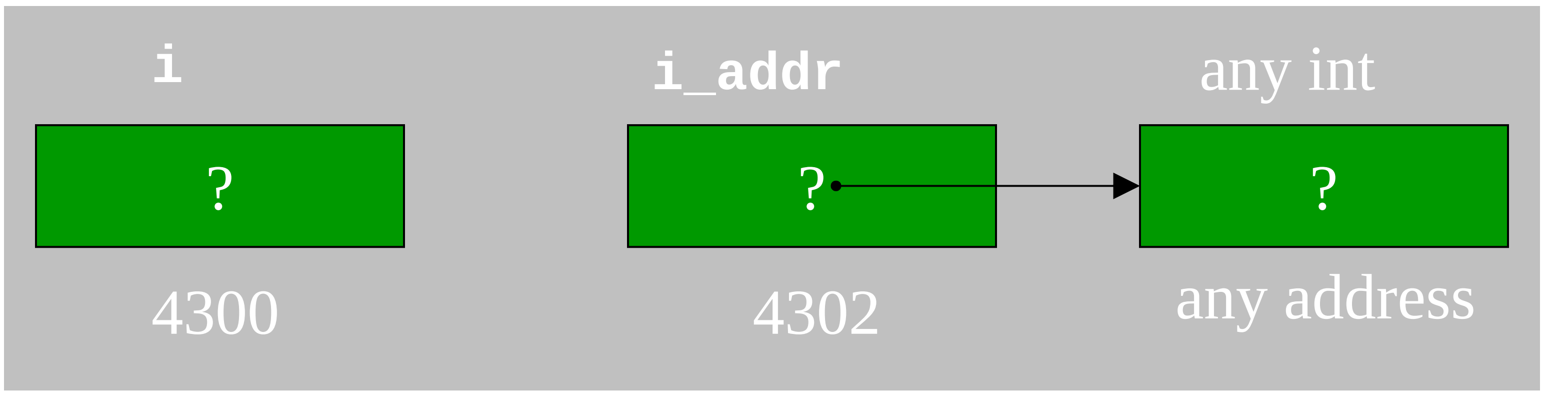
- Pointers are variables that represent an address in memory.
- That location a pointer addresses contains another variable.

```
int i = 5, j = 10;
char c = ‘c’, d = ‘d’;
```

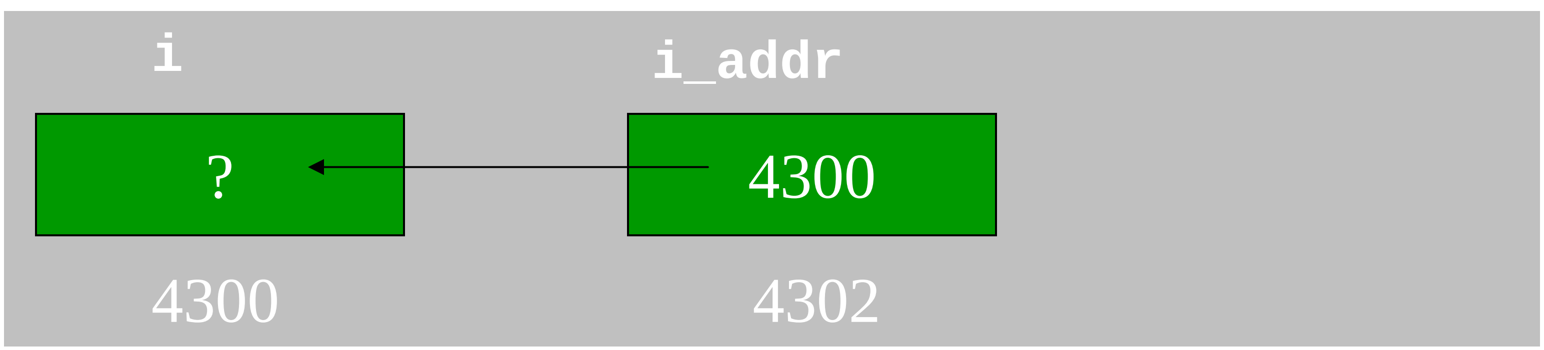


Pointers Made Easy (1)

```
int i;          /* data variable */
int *i_addr;    /* pointer variable */
```

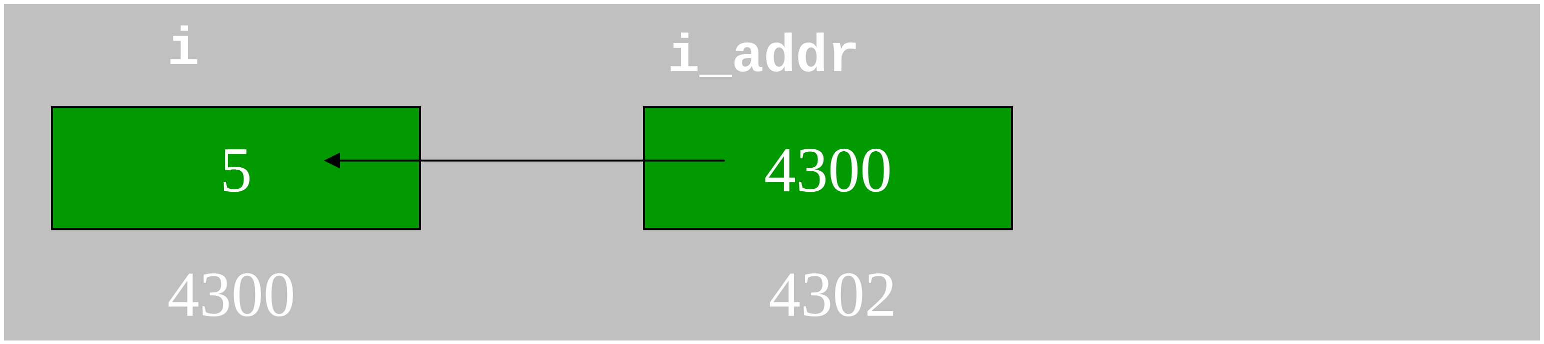



```
i_addr = &i;     /* & = address operator */
```

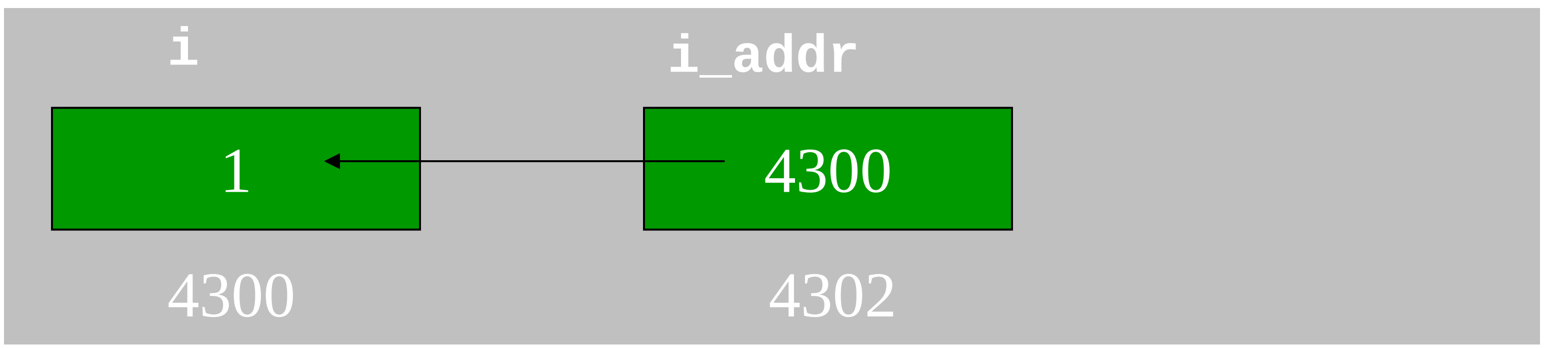


Pointers Made Easy (2)

```
*i_addr = 5;          /* indirection operator */
```




```
int k = *i_addr;/* indirection: k is now 5 */
i = 1;                /* but k is still 5 */
```



Subprograms

- Functions
  - x= f(y)
  - f(x,y)
  - Similar to procedures and subroutines in Fortran or Pascal
- Implemented with LCALL and RET
- Functions are useful for:
  - making code easier to read (structure)
  - reusing code

Program Structure

- Makes code easier to read:

```
init();
while(1) {
  doit();
  if (error) fixup();
}
```

- Reuse blocks of code:

```
move(a,b); move(c,d)
```

Example function

```
PULSEP1.C
#include <reg51.h>
sbit P1_1= P1^1;
void pulseP1B1(void){
 P1_1= 1; P1_1= 0; }
void main(void){
 pulseP1B1();
 pulseP1B1();
}
```

```
 ; FUNCTION pulseP1B1 (BEGIN)
0000 D291              SETB    P1_1
0002 C291              CLR     P1_1
0004 22                RET
; FUNCTION main (BEGIN)
0000 120000   R        LCALL   pulseP1B1
0003 120000   R        LCALL   pulseP1B1
```

Function Parameters

- Function arguments are passed “by value”.
- What is “passing by value”?
  -
- What does this imply?
  -

Example 1: swap_1

```
void swap_1(int a, int b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

Q: Let x=3, y=4,
after swap_1(x,y);
x =? y=?

Example 2

- pass by value
  - `f(x){ x=2 }; //function definition`
  - `f(5);  //does this set 5=2? (no!)`
  - example:

```
; FUNCTION _f (BEGIN)
0000 7F02          MOV     R7,#02H
0002 22            RET
; FUNCTION main (BEGIN)
0000 7F05          MOV     R7,#05H
0002 120000  R     LCALL   _f
```

Output parameters

- So, how do we return something to the caller?
- Pointer parameters

```
void f(char *p){*p= 2}
main() { f(&x); } //sets x=2
```

- Non-void return value:

```
char f(void) {return 2; }
main() { x=f() } //sets x=2;
```

Example 1: swap_2

```
void swap_2(int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

Q: Let x=3, y=4,
after
swap_2(&x,&y);
x =? y=?

Example 2

```
void f(char *p){*p= 2}
main() { f(&x); }      //sets x=2
```

```
 ; FUNCTION _f (BEGIN)
0000 A807           MOV     R0,AR7
0002 7602           MOV     @R0,#02H
0004 22             RET
; FUNCTION main (BEGIN)
0000 7F00     R     MOV     R7,#LOW x
0002 120000   R     LCALL   _f
```

Non-void return value

```
char f(void) {return 2; }
main() { x=f() } //sets x=2;
```

```
 ; FUNCTION f (BEGIN)
0000 7F02              MOV      R7,#02H
0002 22                RET
; FUNCTION main (BEGIN)
0000 120000    R       LCALL    f
0003 8F00      R       MOV      x,R7
```

Modification

- What if we were trying to get to x[i]?

Shift Operator

```
x=0xC2;    // x = 1100 0010
           //shift x left by one bit
x = x<<1; // x = 1000 0100
```

How would we code **`x = x<<3`** in ASM?

Shift Operator

```
x=0xC2;      // x = 1100 0010
             //shift x left by one bit
x = x<<1;    // x = 1000 0100
```

How would we code x = x<<3 in ASM?

```
MOV A,x
CLR C
RLC A
CLR C
RLC A
CLR C
RLC A
MOV x,A
```

Functions and Variable Scope

- Functions:
  - prototype at top
  - pass/return nothing – use type void
  - generally define after main
- Variable Scope
  - may only declare vars at top of program/function
  - globals known everywhere (BAD!)
  - locals known only within their own function
  - static vars keep their value between calls
  - other variables are initialized at each function call

Functions and Variable Scope

```
int blah (int x);
int z;

void main(void){
   int x,y;

   y = 0x5280;
   x = 1;
   z = blah(x);
   z = y;
   x ++;
   z = blah(x);
}

int blah (int v){
   int y = 0;
   static int x=0;

   y ++;
   x ++;

   return v*42;
}
```

Functions and Variable Scope

```
int blah (int x);
int z;

void main(void){
   int x,y;

   y = 0x5280;
   x = 1;
   z = blah(x);
   z = y;
   x ++;
   z = blah(x);
}

int blah (int v){
   int y = 0;
   static int x =0;

   y ++;
   x ++;

   return v*42;
}
```

prototype (global)

local

known

local

Modification

- What if we wanted to return <u>two</u> variables?
- Use pointers

Structured programming

Structured Program Development

- Many early programs were very unstructured (spaghetti-like) due to the use of goto statements and hence difficult to follow and maintain.
- Structured program design is based on the work of Bohm and Jacopini in mid 60s as an improvement on the incomprehensible ‘spaghetti code’.
- It is defined as a method of programming in which programs are constructed with easy-to-follow logic, attempt to use only three basic control structures and are divided into modules.
- Nowadays, all modern higher-level languages are designed to support structured programming.

Characteristics of Structured Programming

- Program is made of small, understandable and manageable modules.
- A module is a part of a program that is dedicated to performing one action. (e.g. each function, subroutine etc. is a kind of module).
- Any program, regardless of its complexity can be formed by three basic elements of control structure:
  - Sequence
  - Selection/Alternation/Decision
  - Iteration/Repetition/Looping
- Each block of code (i.e. sequence of statements) has a single entry point and a single exit point. Thus we can chain sequences of statements together in an orderly fashion.

Code Blocks: Single Entry/Exit

- Sequence of N statements
  - Start as statement 1
  - End at statement N
- Selection structure:
  - Start with If or Select
  - End with End If or End Select
- Repetition structure:
  - While condition Do
  - . . .
  - End While

Advantages of Structured Programming

- The advantages of adopting a structured approach to programming include the following:
  - It makes the program
    - easier to read
    - easier to understand
    - easier to debug
    - easier to maintain
    - more portable
    - easier to reuse
  - A structured programming approach lends itself to team programming.

Flowcharts and Pseudocode

- Structured program languages lend themselves to flowcharts and pseudocode.
- A flowchart is a pictorial representation of a structured program. A flowchart is designed to visually represent the flow of execution through a program.
- Although flowcharts are used quite widely in a variety of situations, they are not often used in algorithms due to ambiguities in their structure. In this course we use pseudocode instead.
- Pseudocode literally means “false code”. It is an English-like, natural language description which concentrates on the logic behind in a program --- not the syntax of a programming language. It is considered as the “first draft” of the actual program.

Structure Programming in Assembly Language

- Assembly language does not formally support structured techniques such as those of high level languages.
- Though the resulted code is often slightly longer, it is possible to implement these structures in assembly language.
- This results in the programs being easier to understand, document, and maintain.

The Concept of Sequence

- Certain events must occur in a particular order - for example, we should get out of bed before showering.
- Other events may occur in any order and do not affect the overall solution.
- In the solution of any problem, it is necessary to decide whether any steps must come before or after other steps.
- Often the effectiveness of a solution may depend on whether or not this happens.
- The same deliberations do not have to occur when the order of steps is not important.

Sequence Structure

- A sequence is a linear structure in which instructions are executed consecutively.
- The most common statement used in high-level languages for this structure is the assignment statement.
  - For example: result = A + B
- To implement this as closely as possible in assembly language:
  - Firstly, a MOV instruction is needed for at least one of the right hand side variables to move it into a register.
  - Then an arithmetic or logical instruction is needed for each operation to be performed.
  - Finally, a MOV is needed to store the calculated result into the variable whose name is on the left-hand side.

Sequence Structure Example

- Pseudocode

  W = (X + Y) * Z

  We assume all the variables W, X, Y and Z are in the 8051's internal memory and have been declared by the EQU directives.

- 8051 ASM code

```
MOV A,X      ;Get X
ADD A,Y      ;Compute X + Y
MOV B,Z      ;Get Z to B
MUL AB       ;Compute (X + Y) * Z
MOV W,A      ;Store result in W
```

  (For simplicity's sake, it is assumed that the product is $\leq$ 255)

Selection Structures

- High-level languages (HLLs) use selection structures such as IF-THEN, IF-THEN-ELSE and SWITCH statements to control block of code execution.
- Selection structures supported by assembly language are very simple, such as testing if the value of a variable (stored in a register) is zero or if the addition of two variables produces a carry or not.
- More complex selection conditions must be translated into a sequence of simple assembly language instructions whose final result will be an equivalent condition test.

Switch Statement

- The SWITCH statement is a handy variation of the IF-THEN-ELSE statement.
- It is used when one statement from many must be chosen as determined by a value.
- Pseudo-code:

```
SWITCH (selector) {
    case value1: Statement 1;
                 break;
    case value2: Statement 2;
                 break;
              . . . . . .
    case valuen: Statement n;
                 break;
    default:     Default statement;
}
```

Example of SWITCH Statement

- e.g. Implement the following pseudo-code in 8051 assembly language. Assume GRADE and SCORE are predefined variables.

```
SWITCH (GRADE) {
  CASE 'A':   SCORE = 90;
              BREAK;
  CASE 'B':   SCORE = 70;
              BREAK;
  CASE 'C':   SCORE = 50;
              BREAK;
  DEFAULT: SCORE = 25;
}
```

Example of SWITCH Statement (Cont.)

8051 assembly code

```
            MOV    A,GRADE            ;Get GRADE to A
CASE0:      CJNE   A,#'A',CASE1       ;If GRADE ≠ 'A' then CASE1
            MOV    SCORE,#90          ;assign SCORE to 90 for case 0
            SJMP   CONTINUE
CASE1:      CJNE   A,#'B',CASE2       ;If GRADE ≠ 'B' then CASE2
            MOV    SCORE,#70          ;assign SCORE to 70 for case 1
            SJMP   CONTINUE
CASE2:      CJNE   A,#'C',DEFAULT     ;If GRADE ≠ 'C' then DEFAULT
            MOV    SCORE,#50          ;assign SCORE to 50 for case 2
            SJMP   CONTINUE
DEFAULT:    MOV    SCORE,#25          ; assign SCORE to 25 for default
CONTINUE:   . . . . . . . .
```

Final advice about coding

- Table lookup may be more efficient than calculation.
- Use only necessary precision; nothing more.
- Use the smallest data type possible; avoid large types such as “float.”
- Read examples in your lecture handout.

For the next lecture

- Download “C for the 8051” from:
  http://ubermensch.org/Computing/8051/
- Review lecture notes.