1)

C and the related C++ language represented the main language for over 80% of embedded programmers (from 2013 survey).

True

False

**Correct**

As such, knowing C well is important for embedded programmers.

2)

C was originally designed for embedded systems.

True

False

**Correct**

C was created in 1972 for mainframe systems; embedded systems didn't really exist in 1972 (processors were too big).

3)

Assembly language is the main language for nearly half of embedded programmers.

True

False

**Correct**

Assembly dominated in early embedded systems days (1980s) due to the need for extremely efficient code, but today's compilers generate efficient code, plus most embedded systems are too big to manage in assembly. Sometimes critical program pieces still need to be written in assembly though.

4)

Fluency in C is rarely useful outside embedded systems.

True

False

**Correct**

C is still used in desktop/server systems. Also, C is the basis for languages like C++, Objective-C, C#, and even Java. And, many people believe that C fluency makes for strong programmers in any language due to better program-execution knowledge.

Table 2.2.1: Data types.

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Minimum size\* | Range | Notes |
| signed char | 8 | −128 to 127 |  |
| unsigned char | 8 | 0 to 255 |
| signed short | 16 | −215 to 215−1 | 216 is 65,536 |
| unsigned short | 16 | 0 to 216−1 |
| signed long | 32 | −231 to 231−1 | 232 is about 4 billion |
| unsigned long | 32 | 0 to 232−1 |
| signed int | N | −2N−1 to 2N−1−1 | *Though commonly used, we avoid these due to undefined width* |
| unsigned int | N | 0 to 2N−1 |

Embedded systems commonly deal with 1-bit data items. C does not have a 1-bit data type, which is unfortunate. Thus, 1-bit items are typically represented using an unsigned char, as in: unsigned char myBitVar.

8-bits is sometimes called a byte.

Following these conventions improves code portability, which is the ability to recompile code for a different microprocessor without undesirable changes in program behavior.

Table 2.2.2: Choosing appropriate type.

|  |  |
| --- | --- |
| Purpose | Variable declaration |
| Store a person's age in years | unsigned char age; // Not < 0, not > 255 |
| Store an airplane's speed | unsigned short speed; // Not < 0, not > 64k |
| Store the remaining joules of energy in a car battery | unsigned long energy; // Not < 0, not > 4Gig |
| Store feet of elevation above/below sea level of land | signed short elevation; // Could be < 0, not > 32K or < -32K. |

1)

How many bits is an unsigned char?

**Correct**

8

char is short for character, but is really just a number, and is commonly used in embedded systems for small numbers, to save scarce memory.

2)

How many bits is a signed char?

**Correct**

8

Same as unsigned char, except the eighth bit is used for the sign. Unsigned char can represent 0-255, while signed -128 to 127.

3)

What is 15 in 8-bit binary?

**Correct**

00001111

8+4+2+1 is 15.

4)

What are the bits stored in memory for unsigned char x = 5?

**Correct**

00000101

101 yields 4+1 or 5.

5)

What are the bits stored in memory for unsigned char x = 199?

**Correct**

11000111

128+64+4+2+1 is 199.

6)

Define a variable "volts" that can range from -100 to +100 (integers only). End with ;.

**Correct**

signed char volts;

signed char can represent -128 to +127.

7)

Define a variable "height" that will hold a human's height in inches (integers only). End with ;.

**Correct**

unsigned char height;

unsigned char can represent 0-255.

8)

Define a variable "birthYear" that will hold the year a person's birth occurred, range is 1. A.D. to today.

**Correct**

unsigned short birthYear;

unsigned short can represent 0 up to about 64,000.

9)

Define a variable "distMoon" that holds the distance in miles that the moon is from the earth on a given day.

**Correct**

unsigned long distMoon;

unsigned short's max is 64,000. Need long, which can store up to about 4 billion.

10)

Define a variable "button" that will indicate whether a button is pressed or not.

**Correct**

unsigned char button;

unsigned char is the closest type to a single bit. The programmer should only assign 1 or 0.

Built-in grouped bits like A and B enable the programmer to treat RIMS' 8 input bits or 8 output bits as an 8-bit decimal number.

The keyword const, short for constant, can precede any variable declaration, as in: const unsigned char i1 = 5;. A constant variable's value cannot be changed by later code, and thus can help to prevent the introduction of future errors.

1)

Set RIMS' first output pin to 1.

**Correct**

B0 = 1;

When setting one bit, use the individual RIMS output variable like B0.

2)

Set RIMS' 8 output bits to decimal 0.

**Correct**

B = 0;

When treating the eight outputs as a decimal number, use B.

3)

Set RIMS' first output to RIMS' last input.

**Correct**

B0 = A7;

Use the single-bit built-in variables when dealing with single bits.

4)

Set RIMS' outputs to RIMS's inputs minus 1, treating each as a decimal number.

**Correct**

B = A - 1;

Use the groupings when treating inputs or outputs as 8-bit decimal numbers.

5)

Set A7 to 1.

**Correct**

Not possible

An input pin cannot be written by a program (what sense would that make?), and instead is read-only.

6)

Set variable x to B7's current value.

**Correct**

x = B7;

Normally a program writes to B7, which is an output. However, the program can also read B7, which yields the previously-written value, just like for any other variable. Think of B7 as a variable whose value is simply shown at the output pin.

7)

Write an expression (without parentheses) that evaluates to true if A5 is 1.

If ( )

**Correct**

A5orA5 == 1

Because A5 is intended to represent a single bit, preferred expressions are A5 to detect 1, and !A5 to detect 0. A5 == 1 is a less-preferred style, as is A5 == 0.

Hexadecimal, or hex, is a base 16 number, where each digit can have the value of 0, 1, ..., 8, 9, A, B, C, D, E, or F.

1)

Set RIMS' first output pin to 1.



CheckShow answer

**Correct**

B0 = 1;

When setting one bit, use the individual RIMS output variable like B0.

2)

Set RIMS' first output pin to 1 and all other pins to 0.



CheckShow answer

**Correct**

B = 0x01;

The intent is to write 00000001, so hex is used. B = 1 would be bad style.

3)

Set RIMS' last output pin to 1 and all other pins to 0.



CheckShow answer

**Correct**

B = 0x80;

A person familiar with hex sees 0x80 as 1000 0000.

4)

Set RIMS' outputs to all 1s.



CheckShow answer

**Correct**

B = 0xFF;

FF is 11111111 in hex. B = 255 would be bad style.

5)

Set RIMS' outputs to 01010101.



CheckShow answer

**Correct**

B = 0x55;

A person familiar with hex sees 0x55 as 0101 0101.

6)

Fill in the blank of this expression to detect that all 8 RIMS inputs are 1s:  
A == \_\_\_\_.



CheckShow answer

**Correct**

0xFF

FF is 11111111. A == 255 would be bad style.

7)

What expression detects that RIMS' first input pin is 1?



CheckShow answer

**Correct**

A0orA0 == 1

A == 0x01 would be wrong because that expression requires all other inputs to be 0, which wasn't required in this question. A0 == 1 is not the preferred style.

8)

What expression detects that A0 is the only RIMS input that is 1?



CheckShow answer

**Correct**

A == 0x01

A1-A7 must be 0s, so comparing with all inputs is required. A == 1 would be bad style.

1)

What B\_ outputs should be set to 1 for case 3? List in ascending order separated by spaces, i.e., B0 B2 ...



CheckShow answer

**Correct**

B0 B2 B3 B5 B6

Those segments are all but the left-side vertical segments, looking like a 3.

2)

To what should B be set for case 3? B = \_\_\_\_; Use uppercase letters for the hex literal.



CheckShow answer

**Correct**

0x6D

0110 is 4+2 is 6, 1101 is 8+4+1 is 13 which is D (10 is A, 11 is B, 12 is C, 13 is D).

* & : bitwise AND — 1 if both bit-operands are 1s.
* | : bitwise OR — 1 if either or both bit-operands are 1s.
* ^ : bitwise XOR — 1 if exactly one of the two bit-operands is 1 (eXclusive OR)
* ~ : bitwise NOT — 1 if the bit-operand is 0; 0 if bit-operand is 1.

1)

00001111 & 10101010



CheckShow answer

**Correct**

00001010

Particular result bit is 1 only if both corresponding operand bits are 1s.

2)

00001111 | 10101010



CheckShow answer

**Correct**

10101111

Particular result bit is 1 if either (or both) of the operand bits is 1.

3)

00001111 ^ 10101010



CheckShow answer

**Correct**

10100101

Particular result bit is 1 if exactly one of the operand bits is 1.

4)

~00001111



CheckShow answer

**Correct**

11110000

Result bit is opposite of operand bit.

5)

Type a statement that sets B's bits to the opposite of A's bits, so if A is 11110000, B will be 00001111. End with ;



CheckShow answer

**Correct**

B = ~A;

~ performs a bitwise NOT operation. Note that !A yields a different result: if A is 11110000, then !A is 00000000 (A is non-zero, so !A will be zero).

Logical operators &&, ||, and ! (there is no logical XOR operator) treat operands as zero (false) or non-zero (true). So while 0x0F & 0xF0 (bitwise AND) evaluates to 0 because each AND of corresponding operand bits evaluates to 0, in contrast 0x0F && 0xF0 (logical AND) evaluates to 1 because both operands are non-zero.

Bitwise operator lookup table

|  |  |  |  |
| --- | --- | --- | --- |
| b | b | b | b |
| & | & | | | | |
| 1 | 0 | 1 | 0 |
| b | 0 | 1 | b |

 A mask is a constant pattern of 0s and 1s, as in 0x0F, used with bitwise operators to manipulate a value.

* *0 & x yields 0* — Used to set a particular bit to 0.
* *1 | x yields 1* — Used to set a particular bit to 1.
* *0 |x yields x*, or *1 & x yields x* — Used to pass a bit through unchanged.

1)

A & 0x03



CheckShow answer

**Correct**

00000011

Particular result bit is 1 only if both corresponding operand bits are 1s.

2)

A | 0xF0



CheckShow answer

**Correct**

11111111

Particular result bit is 1 if either (or both) corresponding operand bits is 1.

3)

A & 0x18



CheckShow answer

**Correct**

00001000

Bit 4 involves 0 and 1, so is 0. Bit 3 involves 1 and 1, so is 1.

4)

A0 && A1



CheckShow answer

**Correct**

00000001

The lack of a single-bit data type in C can be a little confusing. RIMS has single-bit input and output variables like A0 and A1, each stored in an 8-bit variable. Use only logical operators with those variables.

5)

A0 & A1



CheckShow answer

**Correct**

00000001

While the result of using & is the same as using &&, using the bitwise operator on (intended) single-bit variables is*bad practice*.

1)

The expression (A & 0x0F) evaluates to true if any of A3, A2, A1, or A0 is 1.

True

False

**Correct**

For example, if A is 11000011, then 11000011 & 00001111 yields 00000011, which is non-zero so considered true. The expression A3 || A2 || A1 || A0 would also work. Note that the values on A7, A6, A5, and A4 don't matter.

2)

The expression (A && 0xF0) evaluates to true if any of A7, A6, A5, or A4 is 1.

True

False

**Correct**

Logical operator && was used. So 0xF0, which is non-zero, is always considered true. And if *any* bit in A is 1, then A is non-zero and also true.

3)

The expression (A == 0x0F) is a correct way to check if A3, A2, A1, and A0 are all 1s, and A7 through A4 don't matter.

True

False

**Correct**

The expression checks that A3, A2, A1, and A0 are 1s, AND that A7, A6, A5, and A4 are 0s.

4)

The expression (A == 0xF) is a correct way to check if A3, A2, A1, and A0 are all 1s, and A7 through A4 don't matter.

True

False

**Correct**

A is 8 bits, so 0xF has 0s filled in and thus is the same as 0x0F. So the expression checks that A3, A2, A1, and A0 are 1s, AND that A7, A6, A5, and A4 are 0s.

5)

The expression ( (A & 0x0F) == 0x0F ) is a correct way to check if A3, A2, A1, and A0 are all 1s, and A7 through A4 don't matter.

True

False

**Correct**

(A & 0x0F) yields bits 0000A3A2A1A0. Comparing that result with 00001111 is really only comparing the low four bits (the high four bits are always 0s on both sides). Of course, the expression (A3 && A2 && A1 && A0) might be clearer.

6)

The expression ( A == 0xFC ) is a correct way to check if A7 through A2 are all 1s, and A1 and A0 are 0s.

True

False

**Correct**

0xFC is 11111100, and == compares every bit. So for equality, A7 through A2 must be 1s, and A1 and A0 must be 0s.

7)

The expression ( !A1 && !A0 ) is a correct way to check if A7 through A2 are all 1s, and A1 and A0 are 0s.

True

False

**Correct**

While this approach checks that A1 and A0 are both 0s, the approach ignores A7 through A2.

8)

B = A | 0xC0  sets B's bits to A's bits except that B7 and B6 are forced to 1s.

True

False

**Correct**

0xC0 is 11000000.   
1 OR x is 1, so B7 and B6 will be 1s.   
0 OR x is x, so B5 through B0 will be set to A5 through A0.

* *bitpat << amt* : Left shift bitpat by amt positions
* *bitpat >> amt* : Right shift bitpat by amt positions

1)

Given A is 01100111, what is A << 1?



CheckShow answer

**Correct**

11001110

Each bit moves left one place. 0 is shifted into the rightmost bit.

2)

Given A is 01100111, what is A >> 3?



CheckShow answer

**Correct**

00001100

Each bit moves right three places, with the rightmost 1s "falling off". 0 is shifted into the leftmost three bits.

3)

Type a shift expression that results in bit A4 being in the rightmost bit. Type answer in this form: A >> 7



CheckShow answer

**Correct**

A >> 4

Given the original eight bits are A7A6A5A4A3A2A1A0, then A4 must move four places to the right.

4)

Type a single statement (including the ending semicolon) that sets B3..B0 to A7..A4, while setting B7..B4 to 0s. Use a shift operator and no other bitwise operators.



CheckShow answer

**Correct**

B = A >> 4;

Given the original eight bits A7A6A5A4A3A2A1A0, A >> 4 yields 0000A7A6A5A4. Recall that 0s are shifted in.

5)

Suppose A3..A0 hold a 4-bit binary number, and A7..A4 are used for another purpose. Fill in the blank so that this statement sets B to A3..A0 plus 5: B = (\_\_\_\_) + 5;



CheckShow answer

**Correct**

A & 0x0F

The & with 0x0F sets the high four bits to 0s, leaving 0000A3A2A1A0, which is then added with 5. The eight-bit result is written to the eight bits of B.

6)

Suppose A7..A4 hold a 4-bit binary number, and A3..A0 are used for another purpose. Fill in the blank so that this statement sets B to A7..A4 plus 5: B = (\_\_\_\_) + 5;



CheckShow answer

**Correct**

A >> 4

A >> 4 yields 0000A7A6A5A4. That value is added with 5, and the eight-bit result written to the eight bits of B.

7)

Type a statement (including the ending semicolon) that sets B7..B4 to the value in variable num, which is an unsigned char that only holds values from 0 to 15. B3..B0 can be set to 0. End with ;



CheckShow answer

**Correct**

B = num << 4;

num ranges from 00000000 (0) to 00001111 (15). If num's bits are n7n6n5n4n3n2n1n0, then num << 4 yields n3n2n1n00000.

The function uses C's ternary conditional operator (?:). The operator checks the first operand: If non-zero, the expression evaluates to the second operand, else the expression evaluates to the third operand.