

1. [15] This question concerns a C type `set99`, which corresponds to the Pascal type `setof [0..99]`.

a. How would this `set99` be implemented in C so as to minimize its worst-case storage requirements? Assume that `sizeof(char) = 1`, `sizeof(short) = 2`, `sizeof(int) = 4`, and `sizeof(long) = 8`, and express your answer in the form of a typedef.

```
typedef char[13] set99
```

b. Define the set-membership function `int elem(int n, set99 s)`, which returns 1 if `s` contains `n` and 0 otherwise.

```
int elem( int n, set99 s )
{
    if ( 0 <= n && n <= 99 )
        return (s[ n/8 ] >> (n % 8)) & 1; // or s[ n>>3 ] >> (n & 7) & 1
    return 0;
}
```

c. Define the element-deletion function `void delete(int n, set99 s)`, so that after execution of `delete(n s)`, `s` does not contain `n`.

```
void delete( int n, set99 s )
{
    if ( 0 <= n && n <= 99 )
        s[ n/8 ] &= ~( '1' << (n % 8) );
}
```

2. [5] Assuming that

```
sizeof( int ) = 4
```

```
sizeof( double ) = 10
```

predict this program fragment's output:

```
typedef struct { double y; int x; } ArrayDI;
ArrayDI X[2]; cout << &(X[25]) - X;
```

The output:

25

3. [10] You are given the following program fragment in C:

```
int a = 0;
int b = (++a || a++) > (a++ && ++a);
printf( "%d %d", a, b )
```

Assuming that all side-effects take effect immediately, what are the possible outputs?

Whenever the order of operand evaluation is not specified, assume...

a. left operands are evaluated first:

```
b = (++a || a++) > (a++ && ++a) where a = 0
b = (1 || a++) > (a++ && ++a) where a = 1
b = 1 > (a++ && ++a) where a = 1
b = 1 > (1 && ++a) where a = 2
b = 1 > (1 && 3) where a = 3
b = 1 > 1 where a = 3
b = 0 where a = 3
```

So the output is **3 0** .

b. right operands are evaluated first:

```
b = (++a || a++) > (a++ && ++a) where a = 0
b = (++a || a++) > (0 && ++a) where a = 1
b = (++a || a++) > 0 where a = 1
b = (2 || a++) > 0 where a = 2
b = 1 > 0 where a = 2
b = 1 where a = 2
```

So the output is **2 1** .

4. [15] Design a Haskell algebraic type, with instance declarations for Eq, Ord, and Show, for representing a vehicle's speed in either miles per hour or km/hr. Note: 1 mile = 1.61 km.

```

data Speed = MPH Double | KPH Double

toMetric :: Speed -> Double
toMetric (MPH s) = 1.61 * s
toMetric (KPH s) = s

instance Eq Speed where
  s1 == s2 = toMetric s1 == toMetric s2

instance Ord Speed where
  s1 <= s2 = toMetric s1 <= toMetric s2

instance Show Speed where
  show (MPH s) = show s ++ " mi/hr"
  show (KPH s) = show s ++ " km/hr"

```

5. [10] Given the following function definitions (in a Haskell-like notation)

```

f y = 4*y           — in the answers, this f is called f1
g x = 3 * f x

```

predict the value of the expression

```

g (f 2) where f z = z+2   — in the answers, the f defined here is called f0

```

assuming

a. static (i.e., lexical) binding

```

g (f 2)
~ 3 * f1 (f0 2)
~ 3 * 4 * (f0 2)
~ 3 * 4 * (2+2)
~ 3 * 4 * 4
~ 12 * 4
~ 48

```

b. dynamic binding

```

g (f 2)
~ 3 * f0 (f0 2)
~ 3 * f0 (2+2)
~ 3 * f0 4
~ 3 * (4+2)
~ 3 * 6
~ 18

```

6. [10] Given the following C++ program

```

int a = 5;
void cat( int x ){ cout << a; cout << " " << x; cout << " " << x; }
void mouse( int a ){ cat( a+=2 ); }
void main( )
{
    int x = 20;
    mouse( x+=4 );
}

```

a. Predict the output.

5 26 26

b. Replace the definition of void cat(int) with

```

#define cat( x ) { cout << (a); cout << " " << (x); cout << " " << (x); }

```

and again predict the output.

24 26 28

7. [10] Given the following program

```

int k = 12;
procedure bam( int boozle )
{
    boozle := boozle * 2; k := 10;
}
bam( k ); print k;

```

predict the output, assuming arguments are bound to parameters

a. by reference: 10

b. by value-result: 24

8. [10] The following fragmentary program text contains declarations of several procedures with both local and global variables. One of the procedures contains a reference to one of the variables; to answer this question, write the address information that a compiler would generate for that variable reference.

Assume

- a typical stack implementation
- space is allocated for variables in the order in which they are declared, beginning with offset 0
- no requirement that variables be aligned on word boundaries.

Assume that addresses identify bytes, and that the basic data types' space requirements (in bytes) are as follows:

bool 1 **char** 1 **int** 4 **float** 8

The program:

```

program
  i : int ;
  type Rec = record f1 : float [0..10]; f2 : int [0..2] end ;
  y : Rec;
  z : int [1..10];
  j : int ;
  procedure p1
    a : Rec;
    y : float ;
  begin
    ...
  end p1;
  procedure p2
    m : bool[0..99];
    x : float ;
    a : Rec[0..3];
  begin
    ... a[2].f2[2] ...
  end p2;
begin
  ...
end program .

```

First we build a table for procedure p2 showing each variable's size and its offset in its block's activation record.

procedure p2	m	100*1	0-99
	x	8	100-107
	a	4*100	108-507

Because local-variable addressing begins at 0 in each block's activation record, the program's other blocks are irrelevant.

Variable a is an array of Rec; an instance of Rec consists of an array of 11 floats followed by 3 ints, for a total of $11*8 + 3*4 = 100$ bytes.

Hence the offset corresponding to `a[2].f2[2]` would be calculated as follows:

a	108	
a[2]	$108 + 2*\text{sizeof}(\text{Rec}) = 108 + 2*100$	= 308
a[2].f2	$308 + \text{sizeof}(\text{f1}) = 308 + 11*\text{sizeof}(\text{float}) = 308 + 11*8 = 396$	
a[2].f2[2]	$396 + 2*\text{sizeof}(\text{int}) = 396 + 2*4$	= <u>404</u>