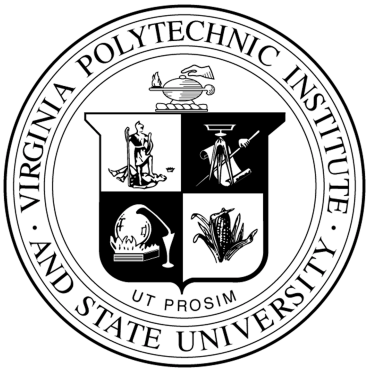


READ THIS NOW!

- Print your name in the space provided below.
- Print your name and ID number on the Opscan form; be sure to code your ID number on the Opscan form. Code **Form B** on the Opscan; code your section **group** number: Barnette 8:00TuTh = 1; McQuain 10:00MWF= 2; or McQuain 12:00MWF = 3.
- Choose the single best answer for each question — some answers may be partially correct. If you mark more than one answer, it will be counted wrong.
- Unless a question involves determining whether given C++ code is syntactically correct, assume that it is valid. The given code has been compiled and tested, except where there are deliberate errors. Unless a question specifically deals with compiler `#include` directives, you should assume the necessary header files have been included.
- Be careful to distinguish integer values from floating point (real) values (containing a decimal point). In questions/answers which require a distinction between integer and real values, integers will be represented without a decimal point, whereas real values will have a decimal point, [1704 (integer), 1704.0 (real)].
- The answers you mark on the Opscan form will be considered your official answers.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- This is a closed-book, closed-notes examination. No calculators or other electronic devices may be used during this examination. You may not discuss (in any form: written, verbal or electronic) the content of this examination with any student who has not taken it. You must return this test form when you complete the examination. Failure to adhere to any of these restrictions is an Honor Code violation.
- There are 34 questions, equally weighted. The maximum score on this test is 100 points.

Do not start the test until instructed to do so!

Print Name (Last, First) Solution

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

signature

I. Class Basics

Assume the following class declaration and implementation:

```
class Flashlight {
private:
   bool  light;   //true - light is on
   int   battery; //0 depleted
public:
   Flashlight ();
   Flashlight (bool state, int charge);
   void on();
   void off();
   bool onoff();
   void recharge (int charge);
   int  power();
};

Flashlight::Flashlight() {
   light   = false; //light off
   battery = 5;     //full charge
}

Flashlight::Flashlight(bool state,
                       int  charge) {
   light   = state;
   battery = charge;
}
```

```
void Flashlight::on() {
   light = true;
}

void Flashlight::off() {
   light = false;
}

bool Flashlight::onoff() {
   return light;
}

void Flashlight::recharge(int charge) {
   battery += charge;
}

int Flashlight::power() {
   return battery;
}
```

Circle the number of the best answer to each question:

#1 How many default constructors does the above class declaration contain?

(1) 1

(2) 2

(3) 2

(4) 4

(5) 0

The "default" constructor is the one that takes 0 parameters.

#2 What does the following statement accomplish: `Flashlight Compact(true, 1);`

Uses the second constructor.

(1) define an instance of the class `Compact`

(2) define an instance named `Compact` of a class `Flashlight` with unknown status

(3) define an instance named `Flashlight` of a class `Compact` with unknown status

(4) define an instance named `Compact` of a class `Flashlight` with its light on and battery minimally charged

(5) define an instance named `Flashlight` of a class `Compact` with its light on and battery minimally charged

(6) None of these

#3 What does the following statement accomplish:

```
Flashlight Keyring;
```

Uses the default constructor.

(1) define an instance of the class `Keyring`

(2) define an instance named `Keyring` of a class `Flashlight` with unknown status

(3) define an instance named `Flashlight` of a class `Keyring` with unknown status

(4) define an instance named `Keyring` of a class `Flashlight` with its light off and battery fully charged

(5) define an instance named `Flashlight` of a class `Keyring` with its light off and battery fully charged

(6) None of these

#4 What do the following statements accomplish:

```
Flashlight Belt(false, 5);
Belt.recharge(-5);
```

(1) instructs the `Flashlight` object `Belt` to fully charge its battery

(2) instructs the `Flashlight` object `Belt` to completely discharge its battery

(3) instructs the `Flashlight` object `Belt` to turn on and stay on until its battery is completely discharged

(4) the statement contains a syntax error

(5) None of these

#5 What do the following statements accomplish:

```
void Blink (Flashlight& hand);

// in main ()
Flashlight hand;
hand.Blink;


void Blink (Flashlight& hand) {
   for (int i=0; i<3; i++)
   {     hand.off();  hand.on();  }
   hand.off();
}
```

(1) causes the `Flashlight` object `hand` to flash three times

(2) causes the `Flashlight` object `hand` to accidentally be left off

(3) causes the `Flashlight` object `hand` be almost depleted of its charge

(4) the statement contains a compilation error

(5) None of these

II. Pointers

#6 What value is printed by the code fragment below?

```
const int SIZE = 10;
int* a; int* b;

a = new int[SIZE];  // assume allocation starts at address 00001000

for (int i =0; i < SIZE; i++)
      a[i] = i;

b = a;
b++;
cout << " b = " << *b << endl;
```

(1) 00001000 (2) 00001004 (3) 0

(4) 1 (5) 10 (6) None of the above

First b points to a[0]. After the increment operation, b points to a[1].

The output statement prints the value of the TARGET of b, which is a[1], and contains 1.

Consider the following code:

```
void resize (int* ray, int then,
             int now);

void main() {
const int SIZE = 10;
int* a;

  a = new int[SIZE];

  for (int i =0; i < SIZE; i++)
      a[i] = i;

  resize(a, SIZE, 2*SIZE);
}
```

```
//resize actual array to dimension now
void resize (int* ray, int then,
             int now)
{
  int *tmp, *p, *q;
  int i;

  p = tmp = new int[now]; //get new array
  for (i=0, q=ray; i<then; i++, p++, q++)
      *p = *q;   //copy from ray to tmp
  delete [] ray; //deallocate ray
  ray = tmp;     //point ray to new
}
```

#7 For the `resize()` function to have its specified effect, which of the following interfaces for `resize()` should be used?

(1) `void resize (int* ray, int then, int now);  //leave as is`

(2) `void resize (int& ray, int then, int now);`

(3) `void resize (int&* ray, int then, int now);`

(4) `void resize (int*& ray, int then, int now);`

(5) `void resize (int* const ray, int then, int now);`

(6) `void resize (const int* ray, int then, int now);`

(7) `void resize (const int* const ray, int then, int now);`

The resize function changes the value of the array pointer so the pointer needs to be passed by reference.

#8 In the code above, how is the dynamic array pointer variable `a` being passed to the `resize()` function?

(1) by value (2) by reference (3) by const reference

(4) as a const pointer (5) as a pointer to a const target (6)) as a const pointer to a const target

Assume the following declarations:

```
const int SIZE = 10;
int x = 0, y[SIZE]={0};

int* a; int* b;
```

Use the responses:

(1) Valid (2) Invalid

for the next 7 questions (#9 - #15). Considering each statement below independently, determine whether each statement would compile (not link) without errors after the statement:

```
a = new int[SIZE];
```

#9 `*a = SIZE;` **(1) Valid**

#10 `a[0] = SIZE;` **(1) Valid**

#11 `delete [] a;` **(1) Valid**

#12 `delete [] y;` **(2) Invalid (y wasn't dynamically allocated)**

#13 `a->[SIZE-1] = SIZE - 1;` **(2) Invalid (syntax error)**

#14 `b = &y[0];` **(1) Valid**

#15 `y = a;` **(2) Invalid (y's a const pointer)**

#16 Identify the logical error that occurs in the code fragment:

```
const int SIZE = 10;

int x = 0;
int b  = (int*) x;

*b = SIZE;
```

(1) Alias pointer exists **(2) Dangling Reference exists**
(3) Illegal memory address reference **(4) Memory garbage exists**
(5) Undefined pointer dereferenced **(6) No logical error occurs**

(b should have been declared as an `int*`.)

#17 Identify the logical error that occurs in the statements:

```
int* w = new int[10];
int* z = new int[10];

w = z;
```

(1) Alias pointer exists
(2) Dangling Reference exists
(3) Illegal memory address reference
(4) Memory garbage exists
(5) Undefined pointer dereferenced
(6) No logical error occurs

Use the responses:

```
   (1) Valid                    (2)  Invalid
```

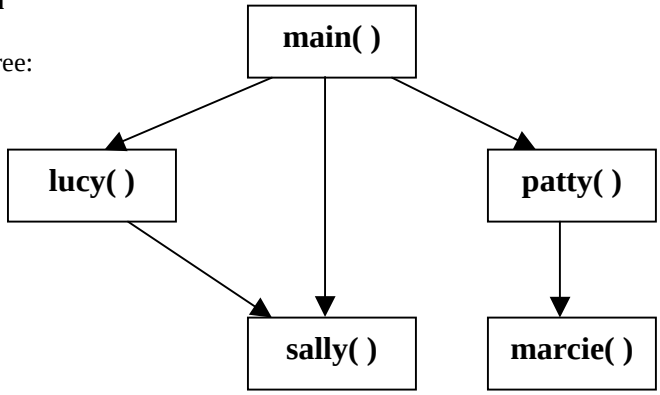
for the next 6 questions (#18 - #23). Considering each numbered question statement in the function below, determine whether each statement would be valid or invalid:

Assume the following function declaration:

```
void fn( int* x) {
  int a[5] = {0, 1, 2, 3, 4};
  const int* b = a;
  b[1] = -1;            //#24: (1)Valid or (2)Invalid ?  Target of b isn't an array
  b = x;                //#25: (1)Valid or (2)Invalid ?
  int* const c = a;
  c[2] = -2;            //#26: (1)Valid or (2)Invalid ?
  c = x;                //#27: (1)Valid or (2)Invalid ?  c is const, can't change it
  const int* const d = a;
  d[3] = -3;            //#28: (1)Valid or (2)Invalid ?  Target of d is const,
  d = x;                //#29: (1)Valid or (2)Invalid ?     and so is d.
}
```

III. Separate Compilation

Consider the function call tree:



Assume that the software system is to be decomposed for compilation into three separate source files `lucy.cpp`, `sally.cpp`, and `patty.cpp`, and accompanying header files of the same names. The function definitions are to be placed in the various cpp files as shown below along with the corresponding code for the files.

`FN definition locations`

Definition for:	Goes in:
`main( )`	`sally.cpp`
`sally( )`	`sally.cpp`
`lucy( )`	`lucy.cpp`
`patty( )`	`patty.cpp`
`marcie( )`	`patty.cpp`

`patty separate compilation unit`

```
//patty.h
void patty ( /* parameters */ );
void marcie( /* parameters */ );
```

```
//patty.cpp
#include "patty.h"

void patty ( /* parameters */ ){

// patty's code

}

void marcie ( /* parameters */ ){

// marcie's code

}
```

`lucy separate compilation unit`

```
//lucy.h
void lucy ( /* parameters */ );
```

```
//lucy.cpp
#include "lucy.h"

void lucy ( /* parameters */ ){

// lucy's code

}
```

`sally separate compilation unit`

```
//sally.h
void sally ( /* parameters */ );
```

```
//sally.cpp
#include "sally.h"

void main() {

   lucy ( /* parameters */ );
   sally( /* parameters */ );
   patty( /* parameters */ );

}

void sally ( /* parameters */ ){

// sally's code

}
```

III. Separate Compilation (continued)

Assume that there are no global type and no constant declarations, (and also no global variables of course). Answer the following questions with respect to the above compilation organization and the goals of achieving information hiding and restricted scope:

#24 Assuming the partial code above was completed and contained no syntax errors, if **only** "`sally.cpp`" is **compiled** (not built) within Microsoft Visual C++, which of the following type of errors would occur:

```
(1) Compilation errors: undeclared identifiers 'lucy', 'patty'
(2) Compilation error: missing main() prototype
(3) Linker Error: multiple identifier redefinitions
(4) No errors would be generated.
```

#25 In how many different files (source and header) should the `#include "sally.h"` directive occur?

```
(1) 1       (2) 2       (3) 3       (4) 4
(5) 5       (6) 6       (7) 7       (8) 0
```

#26 In order to prevent possible linker errors, which of the following actions should be taken:

```
(1) Move main() to a separate compilation unit: main.cpp & main.h
(2) Surround each header file contents with compiler directives (#ifndef UNIT_H,
    #define UNIT_H, #endif) where UNIT is replaced by the file name.
(3) Combine all functions into one cpp file to achieve faster re-compilations.
(4) #include all header files in every .cpp source file.
```

#27 In addition to the include directives listed above, where else should "`sally.h`" be included?

```
(1) lucy.h        (2) sally.h       (3) lucy.cpp
(4) patty.h       (5) patty.cpp     (6) nowhere else
```

#28 In addition to the include directives listed above, where else should "`lucy.h`" be included?

```
(1) lucy.h        (2) sally.h       (3) sally.cpp
(4) patty.h       (5) patty.cpp     (6) nowhere else
```

#29 In addition to the include directives listed above, where else should "`patty.h`" be included?

```
(1) lucy.h        (2) sally.h       (3) sally.cpp
(4) patty.h       (5) patty.cpp     (6) nowhere else
```

#30 Which of the following prototypes should be moved from its unit header file to the unit source file?

```
(1) void patty ( /* parameters */ );      (2) void marcie( /* parameters */ );

(3) void lucy  ( /* parameters */ );      (4) void sally ( /* parameters */ );
```

IV. Design Representation

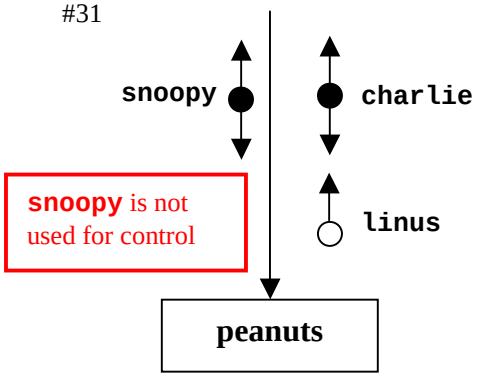
Consider the following incomplete function call and function heading code below:

```
int   snoopy, linus[100];
bool  charlie;

peanuts(charlie, snoopy, linus);
if (charlie)
      //code under control of if
```

```
void peanuts(bool& charlie, int& snoopy, const int linus[]) {

      if (charlie)
            //code under control of if
```

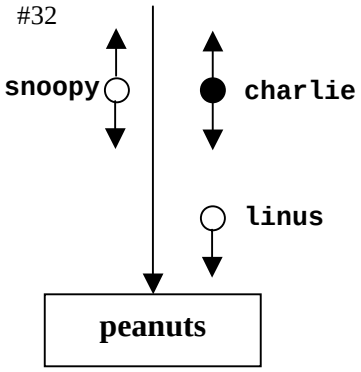
Do not make any assumption about variables that are not shown on the chart. Which of the following structure chart diagrams for `peanuts()` below correctly models the code segment above, (more than 1 may be a valid model)?

#31



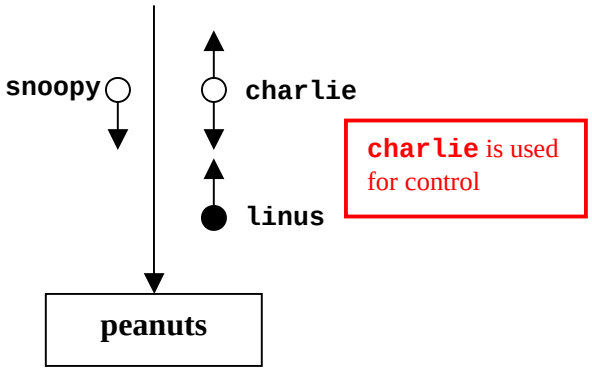
(1) – correct
(2) – incorrect

#32



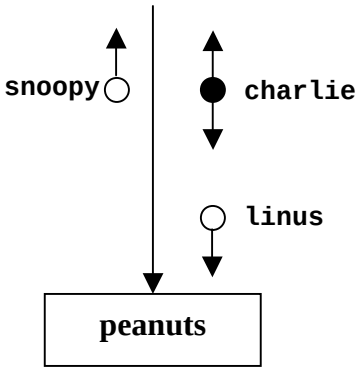
(1) – correct
(2) – incorrect

#33



(1) – correct
(2) – incorrect

#34



(1) – correct
(2) – incorrect