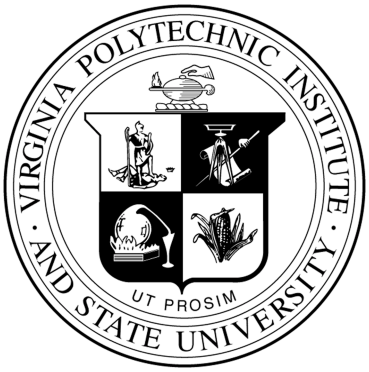


READ THIS NOW!

- 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 A** on the Opscan.
- 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 33 questions, equally weighted. The maximum score on this test is 100 points.

Do not start the test until instructed to do so!

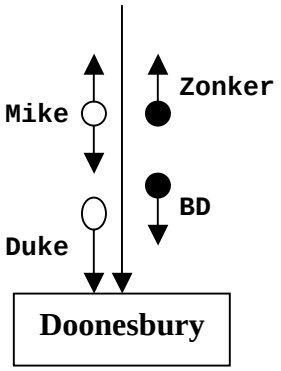
Print Name (Last, First) **Solution**

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

N. D. Barnette
signature

I. Design Representation

Consider the following partial Structure Chart diagram below:



Analysis:
`Zonker` is a control output parameter so it must be passed by reference (or be the return value from the function)
`Mike` is an input/output parameter, so it must be passed by reference (and cannot be the return value from the function)
`Duke` is an input parameter; that means Duke does not need to be passed by reference (and so should not be)
`BD` is a control input parameter; that means BD does not need to be passed by reference (and so should not be)

Given those observations, the results below follow.

KEY

Do not make any assumption about variables that are not shown on the chart. Given the following variables definitions:

```
bool   BD,   Zonker;
int    Mike, Duke;
```

Which of the following incomplete function calls and function heading code below for `Doonesbury()` correctly models the diagram above, (more than 1 may be a valid model:

#1
```
Doonesbury(Mike, Zonker, Duke, BD);
if (Zonker)
   //code under control of if
```
```
void Doonesbury(int& Mike, bool& Zonker,
          const int& Duke, bool  BD) {
   if (BD)
       //code under control of if
```

(1) – correct
(2) – incorrect

#2
```
Doonesbury(Mike, Zonker, Duke, BD);
if (Mike > 0)
   //code under control of if
```
```
void Doonesbury(int& Mike, bool Zonker,
          int  Duke, const bool&  BD) {
   if (Mike < 0)
       //code under control of if
```

(1) – correct
(2) – incorrect

#3
```
Doonesbury(&Mike, &Zonker, Duke, BD);
if (!Zonker)
   //code under control of if
```
```
void Doonesbury(int* Mike, bool* Zonker,
           int Duke, const bool&  BD) {
   if (!BD)
       //code under control of if
```

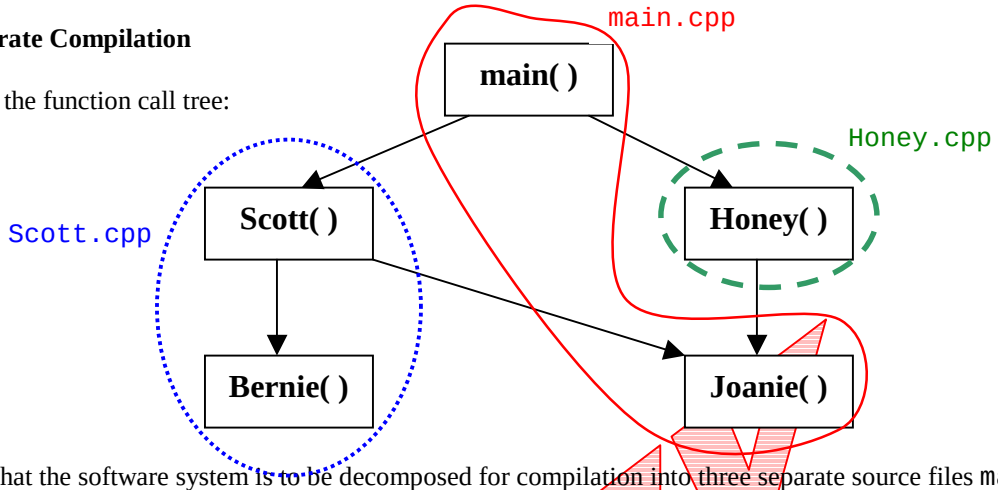
(1) – correct
(2) – incorrect

#4
```
Doonesbury(*Mike, *Zonker, Duke, BD);
if (Zonker == NULL)
   //code under control of if
```
```
void Doonesbury(int& Mike, bool& Zonker,
       const int Duke, const bool  BD) {
   if (BD != NULL)
       //code under control of if
```

(1) – correct
(2) – incorrect

II. Separate Compilation

Consider the function call tree:



Assume that the software system is to be decomposed for compilation into three separate source files `main.cpp`, `Scott.cpp`, and `Honey.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( )`	`main.cpp`
`Joanie( )`	`main.cpp`
`Scott( )`	`Scott.cpp`
`Bernie( )`	`Scott.cpp`
`Honey( )`	`Honey.cpp`

`Scott separate compilation unit`

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

```
//Scott.cpp
#include "Scott.h"
void Bernie( /* parameters */ );

void Scott ( /* parameters */ ){
// Scott's code
   Bernie();
   Joanie();
}

void Bernie ( /* parameters */ ){
// Bernie's code

}
```

`Honey separate compilation unit`

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

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

void Honey ( /* parameters */ ){
// Honey's code
   Joanie ( /* parameters */ );

}
```

`main separate compilation unit`

```
//main.h
/* main declarations */
```

```
//main.cpp
#include "main.h"
void Joanie ( /* parameters */ );

void main() {

   Scott ( /* parameters */ );
   Honey ( /* parameters */ );

}

void Joanie ( /* parameters */ ){
// Joanie's code

}
```

II. 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:

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

(1) Compilation errors: missing Honey() prototype
(2) Compilation error: undeclared identifiers 'Joanie'
(3) Linker Error: missing main function.
(4) No errors would be generated.

Joanie()'s missing prototype generates a undeclared identifier error.

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

(1) `void Honey ( /* parameters */ );`
(2) `void Bernie( /* parameters */ );`
(3) `void Scott ( /* parameters */ );`
(4) `void Joanie ( /* parameters */ );`

KEY

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

(1) main.h ***=0.5*** **(2) main.cpp** (3) Scott.cpp
(4) Scott.h (5) Honey.cpp (6) nowhere else

Honey()'s prototype must be included in main so that it can be called.

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

(1) main.h ***=0.5*** **(2) main.cpp** (3) Scott.cpp
(4) Honey.h (5) Honey.cpp (6) nowhere else

Scott()'s prototype must be included in main so that it can be called.

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

(1) main.h (2) main.cpp **(3) Honey.cpp**
(4) Honey.h (5) Scott.h ***=0.5*** (6) nowhere else

Since Honey() calls Joanie() the header for the file containing Joanie()'s prototype must be included in main so that it can be called.

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

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

Since Honey(), & Scott() calls Joanie() the header for the file containing Joanie()'s prototype must be included, also in main so that it can be compiled.

#11 The name of the linker program that is invoked automatically by the Microsoft Visual C++ development environment is :

(1) cl
(2) ld
(3) ln
(4) link
(5) none of the above

See slide 2.3 of the notes

III. Pointers

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 (#12 - #18). Considering each statement below **independently**, determine whether each statement would compile (not link) without errors after the statement:

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

#12
`b = &y[SIZE];` ***(1) Valid***

#13
`*a = *y;` ***(1) Valid***

#14
`(*a).[SIZE-1] = (*y).[SIZE-1];` ***(2) Invalid*** *a and y are NOT pointers to structs or objects*

#15
`&y = &a;` ***(2) Invalid*** *(address operator & is NOT the dereference * operator)*

#16
`y = NULL;` ***(2) Invalid*** *(Array pointers are constant)*

#17
`a = new int[SIZE];` ***(1) Valid***

#18
`y = new int[SIZE];` ***(2) Invalid*** *(Array pointers are constant)*

KEY

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

```
char *p = new char[5];
strcpy(p, "CPP");

p = &p[0];
```

(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

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

```
char *q = new char[5];
strcpy(q, "BSS");
char *r;

r = &q[0];
```

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

#21 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 = b + SIZE;
cout << " b = " << *b << endl;
```

b == &a[0] and b+1== &a[1], thus b+10 == &a[10] which is beyond the bounds of the array,

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

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

Consider the following code:

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

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

  a = new int[SIZE];

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

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

```
//resize actual array to dimension now
void resize (const 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 [SIZE] ray; //deallocate ray
  ray = tmp;         //point ray to new
}
```

#22 In the code above, how is the dynamic array int 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
(7) none of the above *(2) and (5) == 0.5 credit*

#23 Which of the following statements best describe the result/effect of the call to the `resize()` function from the `main()` function?

(1) the resize() function will correctly allocate a new array, copy the old contents of a into it, remove the memory previously allocated to a and reassign a to point to the new array.

(2) the call to the resize function will result in an array bounds violation when array a is reassigned to the new allocated array.

(3) the call to the resize function will result in an array bounds violation when array a's contents is copied to the new allocated array. *== 0.5 credit*

(4) the call to the resize function will result in the newly allocated array containing locations that have not been initialized.

(5) none of the above

Although the initialization of the new (smaller) array to the old array's contents will access beyond the bounds of the new array, C/C++ does **not** report array bounds errors.

Use the responses:

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

for the next 6 questions (#24 - #29). Considering each numbered question statement in the function below separately, determine whether each statement would be valid or invalid:

Assume the following function declaration:

```
void fn( char* const a) {
  char s[3] = {'a', 'b', 'c'};
  const char* const b = s;
  b = a;                //#24: (1)Valid or (2)Invalid ?
  *b = 'b';             //#25: (1)Valid or (2)Invalid ?
  const char* c = s;
  c = a;                //#26: (1)Valid or (2)Invalid ?
  *c = 'c';             //#27: (1)Valid or (2)Invalid ?
  char* const d = a;
  d = a;                //#28: (1)Valid or (2)Invalid ?
  *d = 'd';             //#29: (1)Valid or (2)Invalid ?
}
```

KEY

b is a constant pointer to a constant target neither it nor its target can be re-assigned.

c is a pointer to a constant, its target can NOT be re-assigned.

d is a constant pointer to a char target thus its pointer value can NOT be re-assigned.

IV. Class Basics

Assume the following class declaration and implementation:

```
class ShotGun {
private:
   bool  safety;   //true – cannot fire
   int   rounds;   //number of shots
public:
   ShotGun(bool safe=true, int ammo=0);
   void safetyon();
   void safetyoff();
   bool ready();
   void load (int shells);
   void eject(int shells);
   int  bullets();
};

ShotGun:: ShotGun (bool safe, int ammo)
{
   safety = safe;
   rounds = ammo;
}

void ShotGun:: safetyon () {
   safety = true;
}

void ShotGun:: safetyoff () {
   safety = false;
}

bool ShotGun:: ready () {
   return((!safety) && (rounds > 0));
}

void ShotGun:: load (int shells) {
   rounds += shells;
}

void ShotGun:: eject (int shells) {
   rounds -= shells;
}

int ShotGun:: bullets () {
   return rounds;
}
```

KEY

Circle the number of the best answer to each question:

#30 What does the following statement accomplish: `ShotGun Browning(false, 6);`

(1) define an instance of the class `Browning`.

(2) define an instance named `Browning` of a class `ShotGun` with unknown status.

(3) define an instance named `ShotGun` of a class `Browning` with unknown status.

(4) define an instance named `ShotGun` of a class `Browning` with 6 possible shots and ready to fire.

(5) define an instance named `Browning` of a class `ShotGun` with 6 possible shots and ready to fire.

(6) None of these

Invokes the parameterized constructor for the ShotGun class.

#31 What does the following statement accomplish: `ShotGun Remington;`

(1) define an instance of the class `Remington`.

(2) define an instance named `Remington` of a class `ShotGun` with unknown status.

(3) define an instance named `ShotGun` of a class `Remington` with unknown status.

(4) define an instance named `ShotGun` of a class `Remington` with 0 possible shots and Not ready to fire.

(5) define an instance named Remington of a class ShotGun with 0 possible shots and Not ready to fire.

(6) None of these

Invokes the default constructor for the ShotGun class.

#32 How many of the member functions in the `ShotGun` class should have been declared as `const` member functions?:

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

Only the member functions ready() and bullets do not change data members.

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

(1) 1
(2) 2
(3) 3
(4) 4
(5) 0

A correct class specification, (of which ShotGun is an example), can have only one default constructor.