

## Introduction to pointer types

1. Assume that the following valid variable declarations have been made:

```
int a = 4;
int b = 8;
double d = 2.1;
int* ptr1;
int* ptr2;
int** ptr3;
```

For each expression, if it is valid, identify the type the expression evaluates to otherwise mark it as invalid:

	type
5;	int
a;	int
&a;	Address of an int: int*
*a;	invalid
ptr1;	int*
*ptr1;	int
*ptr1 + *ptr2;	int + int: int
*ptr2 + d;	int + double: double
ptr3;	Address of an address - int**
&ptr1;	int**
&ptr3;	Address of, an address of, an address of an int: int***
**ptr3;	int
*ptr3;	Address of an int - int*
*ptr1 + 10;	int
**ptr2;	invalid
&ptr2;	Address of an address of an int* - int **
**ptr3 + *ptr1;	int + int: int
ptr1 + 1;	int* + int: int*

2. Assume that the following valid variable declarations have been made:

```
int a = 4;
int b = 8;
double d = 2.1;
int* ptr1;
int* ptr2;
int** ptr3;
```

For each of the following assignment statements, indicate the type of the variable on the left hand side (LHS) of the assignment operator (=), the type the expression on the right hand side (RHS) of the assignment operator evaluates to and whether the assignment is valid. To be valid, the type on the LHS must = the type on the RHS.

	Type LHS	Type RHS	Valid?
*ptr2 = &b;	Value pointed to by an int*: <b>int</b>	Address of int: <b>int*</b>	n
a = **ptr3 + 5;	<b>int</b>	int + int: <b>int</b>	y
**ptr3 = **ptr3 + *ptr1;	Value pointed to by an int*, pointed to by an int**: <b>int</b>	int + int: <b>int</b>	y
ptr3 = &a;	<b>int**</b>	Address of int: <b>int*</b>	n
ptr3 = &ptr2;	<b>int**</b>	Address of an, address of an int*: <b>int**</b>	Y
ptr2 = &a;	<b>int*</b>	Address of an int: <b>int*</b>	Y
*ptr3 = ptr1;	Value pointed to by an int**= Address of int: <b>int*</b>	<b>int*</b>	y
ptr1 = &d;	<b>int*</b>	<b>double*</b>	n
d = *ptr1 + **ptr3;	<b>double</b>	int + int: <b>int</b>	y
<u>*ptr1 = **ptr3 + d;</u>	<u><b>int</b></u>	<u>int + double:</u> <u><b>double</b></u>	<u>y</u>

3. Assume that the following valid variable declarations have been made:

```
int a = 4;
int b = 8;
double d = 2.1;
int* ptr1;
int* ptr2
int** ptr3;
```

Assume that the following valid statements are executed one after another in the given sequence. Update the values of the variables that change after each statement has executed in the trace table provided for you.

TIP: Draw a trace diagram to help you keep track

ptr1 = &a;

ptr2 = &b;

ptr3 = &ptr2;

\*ptr1 = 5;

\*\*ptr3 = 10;

\*ptr3 = ptr1;

\*ptr2 = 12;

\*\*ptr3 = \*ptr2 + \*\*ptr3;

b = 20;

ptr1 = &b;

ptr3 = &ptr1;

a = (\*ptr1)++; **//postfix!**

ptr2 = \*ptr3;

\*ptr2 = 1;

ptr2 = &a;

\*ptr2 = (\*\*ptr3)++;

a = ++(\*ptr1);

\*\*ptr3 = 50;

a	b
4	8
5	
	10
12	
24	
	20
20	21
	1
1	2
3	3
	50

#### 4. Consider the program below.

```
#include <stdio.h>

void foo(int* x, int** y);

int main( void ) {
    int a = 10;
    int b = 11;
    int *c;
    int **d;

    c = &b;
    d = &c;

    foo(a,b);
    printf("a: %d, b: %d\n", a, b);

    foo(&a, &b);
    printf("a: %d, b: %d\n", a, b);

    foo(&a, &c);
    printf("a: %d, b: %d\n", a, b);

    foo(c, d);
    printf("a: %d, b: %d\n", a, b);

    foo(c, &c);
    printf("a: %d, b: %d\n", a, b);

    foo(&c, &d);
    printf("a: %d, b: %d\n", a, b);

    foo(&b, d);
    printf("a: %d, b: %d\n", a, b);

    return 0;
}

void foo(int* x, int** y) {
    (*x)++;
    (**y)++;
}
```

\*somepoint = move forward  
&somevar, &somepoint = move backward

context: tracing table

*p.s: i summed up the whole thing in a tracing table, through the successive use of "\*" and "&" where*

*// d is int\*\* meaning it will take d to go through two addresses in order to reach an integer value*

*// c is int\* meaning it will take c to go through one address in order to reach an integer value*

*// a and b are int values*

*// int\* x can be any int\* values (meaning any pointers that needs to go through only one address to reach an integer)*

*// int\*\* y can be any int\*\* values (meaning any pointers that needs to go through only one address to reach an integer)*

- Given the function prototype for `foo` and the variable declarations and initializations within `main`, which are valid calls to `foo` shown within `main`? TIP: determine the type of the argument being passed in the function call and determine if it matches the signature of `foo` (the types of the arguments `foo` is expecting)  
Cross out any invalid calls and the print statement after an invalid call.
- Once you have eliminated the invalid function calls within `main`, trace the program with the remaining function calls?  
a: 11, b: 12  
a: 11, b: 14  
a: 11, b: 16  
a: 11, b: 18

5. Consider the following program.

```
#include <stdio.h>

void swap(int x, int y);
void swap_ptrs(int* x, int* y);

int main( void ) {
    int a = 10;
    int b = 20;

    printf("before swap - a: %d, b: %d\n", a, b);
    swap(a, b);
    printf("after swap - a: %d, b: %d\n", a, b);

    printf("before swap_ptrs - a: %d, b: %d\n", a, b);
    swap_ptrs(&a, &b);
    printf("after swap_ptrs - a: %d, b: %d\n", a, b);
    return 0;
}

void swap(int x, int y) {
    int tmp = x;
    x = y;
    y = tmp;

    printf("in swap - x: %d, y: %d\n", x, y);
}

void swap_ptrs(int* x, int* y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;

    printf("in swap_ptrs - *x: %d, *y: %d\n", *x, *y);
}
```

a) What is the output?

```
before swap - a: 10, b: 20
in foo - x: 20, y: 10
after swap - a: 10, b: 20
before swap_ptrs - a: 10, b: 20
in swap_ptrs - *x: 20, *y: 10
after swap_ptrs - a: 20, b: 10
```

b) What is different between the 2 functions `swap` and `swap_ptrs`?

Swap of variable values is seen in main with `swap_ptrs` but not with `swap`