Tracing and Visualizing C Programs Using C Coding Tutor: Learning Activities for CCPROG1

Preliminary Task: please click this link for Check-up Quiz #3 and answer a simple non-graded check-up quiz on pointers. The form is only accessible to a DLSU domain; please log-in to your DLSU account in case you cannot access it the first time around. The quiz assumes that you have already covered the topic on pointers in your CCPROG1 class. The objective of the quiz is to determine possible misconceptions which we intend to clarify and remediate as part of the learning activities. The answers to Quiz #3 can be found in the footnote on the last page, i.e., page 12 of this document.

MODULE 4: Tracing and Visualizing Pointers

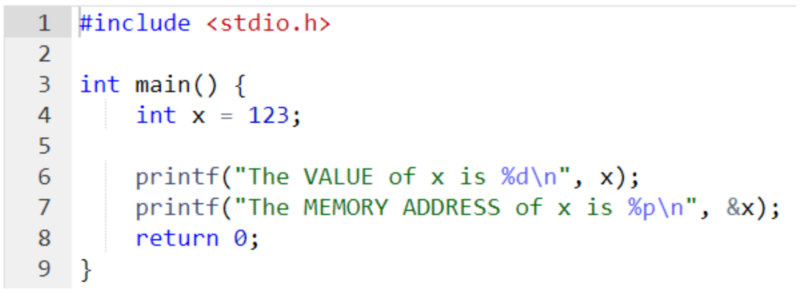
It is assumed that you have covered the topic on pointers in your CCPROG1 class. For this module, we will visualize what happens when we declare a pointer variable (from hereon, referred to shortly as pointer), and when the value of a pointer variable as function parameter.

Learning Activity 4.1: Displaying/printing the memory address of an integer variable

Learning Outcome: The student can access and display a variable's memory address via the address-of operator **&**

Instructions:

1. Encode the source code shown in Figure 4.1. Or if you want to omit the encoding step, you may simply click this link to open a browser window with a pre-encoded program.



```
#include <stdio.h>

int main() {
    int x = 123;

    printf("The VALUE of x is %d\n", x);
    printf("The MEMORY ADDRESS of x is %p\n", &x);
    return 0;
}
```

Figure 4.1: Access the memory address of a variable using the address-of operator &

IMPORTANT: please recall that each variable declared or defined in a C program has the following attributes

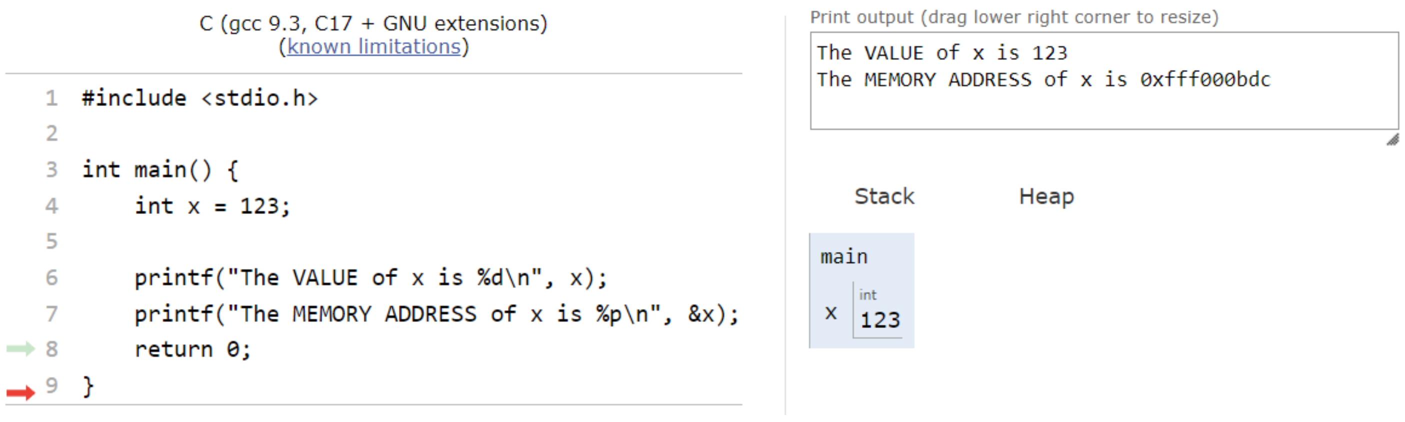
- name of the variable (for example, x in the program above)
- value of the variable (for example, 123 in the program above)
- pointer or memory address of a variable (for example, &x in the program above)

The **&** (ampersand) is the symbol for the **address-of** operator. It is a unary operator, i.e., it requires only one operand which is the name of a variable. The **&** operator should always be written before the name of a variable. The address-of operation will produce as a result the **memory address** of the variable. The **%p** is the formatting symbol for a pointer value.

REMEMBER: we use the term **pointer** to mean the same thing as memory address.

2. Let’s see a visualization of the program. Press “Visualize Execution” button, and press the “Next” button several times until you reach the last step. Make sure to keenly observe what happens on the screen each time you press the “Next” button.

   Figure 4.2 shows the visualization after the last step.


C (gcc 9.3, C17 + GNU extensions)
(known limitations)
Print output (drag lower right corner to resize)
Stack
Heap
main
x
int
123

```c
#include <stdio.h>

int main() {
    int x = 123;

    printf("The VALUE of x is %d\n", x);
    printf("The MEMORY ADDRESS of x is %p\n", &x);
    return 0;
}
```

```
The VALUE of x is 123
The MEMORY ADDRESS of x is 0xfff000bdc
```

Figure 4.2: The memory address of x is indicated as 0xfff000bdc (hexadecimal number)

IMPORTANT: In general, the memory address WILL be different (i.e., the value is not the same) if you compile and run the program on another platform (i.e., different computer, operating system, compiler). To check, try to compile and run the same program in DEVCPP or in VSC IDE installed on your computer. You will notice that the memory address of x is not the same with what we got via C Tutor. The actual value of the memory address itself is not important. What is important for us to know is that there will always be an associated memory address for each variable.

If you are trying to visualize and understand a program with pointers, it is suggested that you draw something similar to Figure 4.3 below. Imagine the rectangular area to be the memory space that contains the value. There are two ways to access this memory space; the easier way is via a DIRECT access using the variable name, and the other is via an INDIRECT access using the memory address (pointer). More details about INDIRECT access will be provided in the succeeding learning activities.

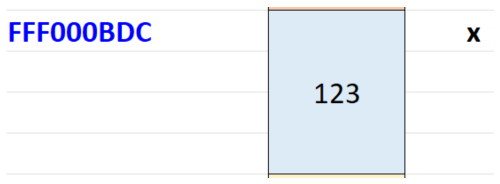


Figure 4.3: Visualization of the memory address (FFF000BDC), value (123) and name (x) of a variable

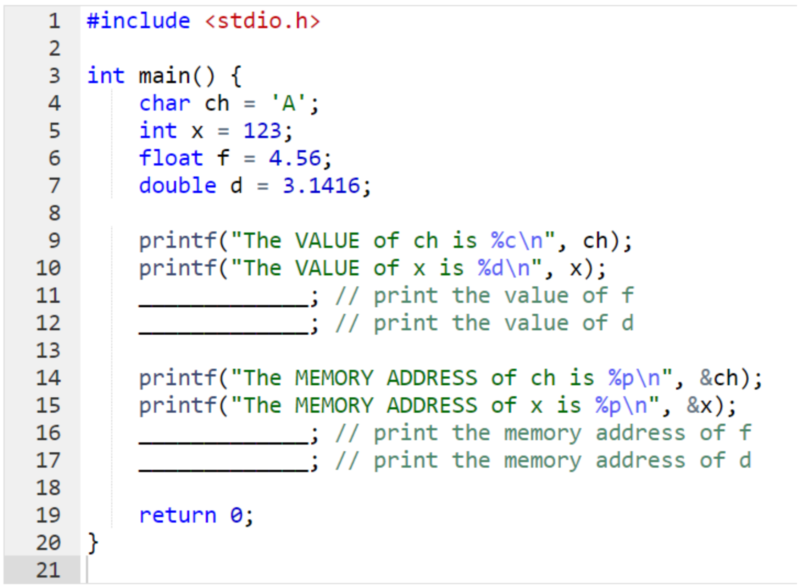
Learning Activity 4.2: Displaying/printing the memory addresses of variables of different data types

Let’s extend the previous activity by examining the addresses of variables with data types other than int.

Learning Outcome: The student can access and display a variable’s memory address via the address-of operator &

Instructions:

1. Encode and supply the missing printf statements in the source code shown in Figure 4.4. Or if you want to omit the encoding step, you may simply click this link to open a browser window with a pre-encoded program.



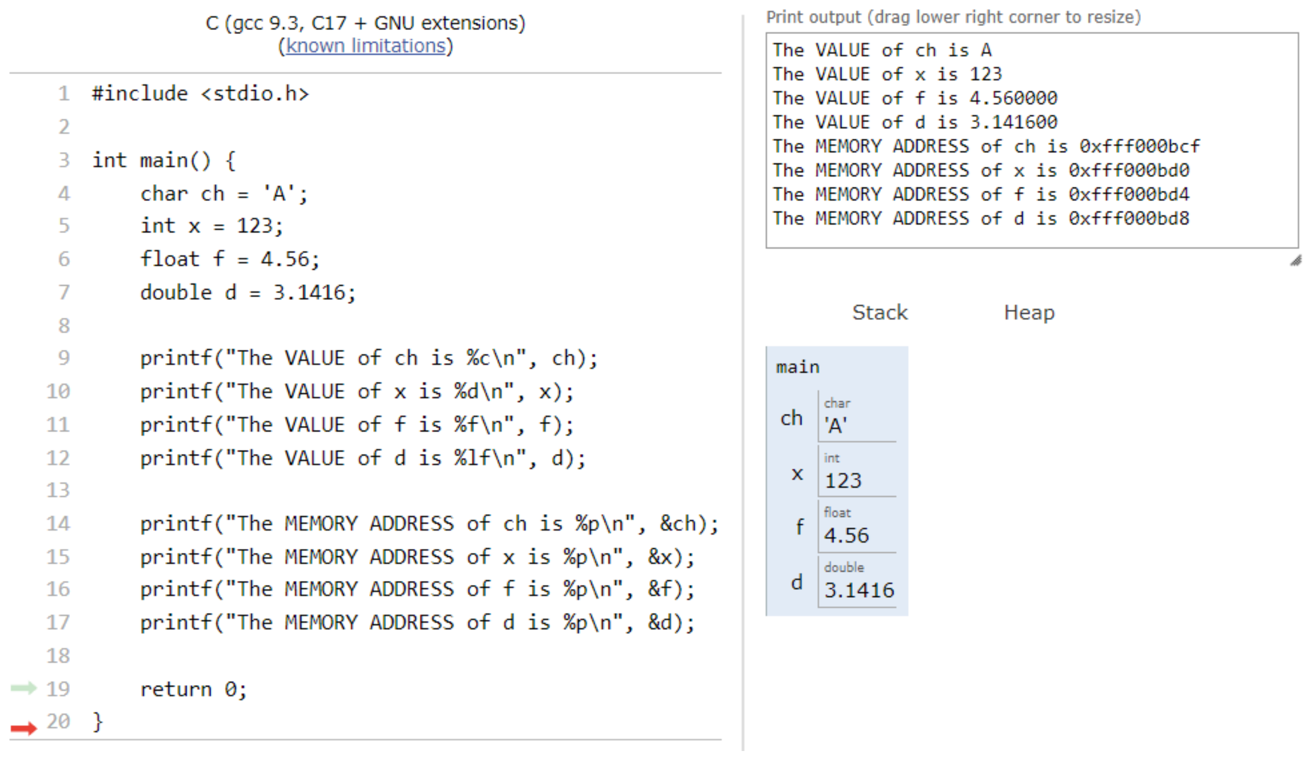
```c
#include <stdio.h>

int main() {
    char ch = 'A';
    int x = 123;
    float f = 4.56;
    double d = 3.1416;

    printf("The VALUE of ch is %c\n", ch);
    printf("The VALUE of x is %d\n", x);
    _____________; // print the value of f
    _____________; // print the value of d

    printf("The MEMORY ADDRESS of ch is %p\n", &ch);
    printf("The MEMORY ADDRESS of x is %p\n", &x);
    _____________; // print the memory address of f
    _____________; // print the memory address of d

    return 0;
}

```

Figure 4.4: Supply the missing printf statements

2. Once the codes are complete, visualize and trace the program to completion by pressing the “Next” button several times. What addresses did you get?

   Draw a visualization of the memory addresses, values and names similar to Figure 4.3 of the previous learning activity. **Please DO NOT proceed to the next page unless you’re done with your drawing.**

The memory addresses that I got from my program are shown in the print output box of Figure 4.5 below.


C (gcc 9.3, C17 + GNU extensions)
(known limitations)
Print output (drag lower right corner to resize)
Stack
Heap
main
ch
char
'A'
x
int
123
f
float
4.56
d
double
3.1416

```c
#include <stdio.h>

int main() {
    char ch = 'A';
    int x = 123;
    float f = 4.56;
    double d = 3.1416;

    printf("The VALUE of ch is %c\n", ch);
    printf("The VALUE of x is %d\n", x);
    printf("The VALUE of f is %f\n", f);
    printf("The VALUE of d is %lf\n", d);

    printf("The MEMORY ADDRESS of ch is %p\n", &ch);
    printf("The MEMORY ADDRESS of x is %p\n", &x);
    printf("The MEMORY ADDRESS of f is %p\n", &f);
    printf("The MEMORY ADDRESS of d is %p\n", &d);

    return 0;
}
```

```
The VALUE of ch is A
The VALUE of x is 123
The VALUE of f is 4.560000
The VALUE of d is 3.141600
The MEMORY ADDRESS of ch is 0xfff000bcf
The MEMORY ADDRESS of x is 0xfff000bd0
The MEMORY ADDRESS of f is 0xfff000bd4
The MEMORY ADDRESS of d is 0xfff000bd8
```

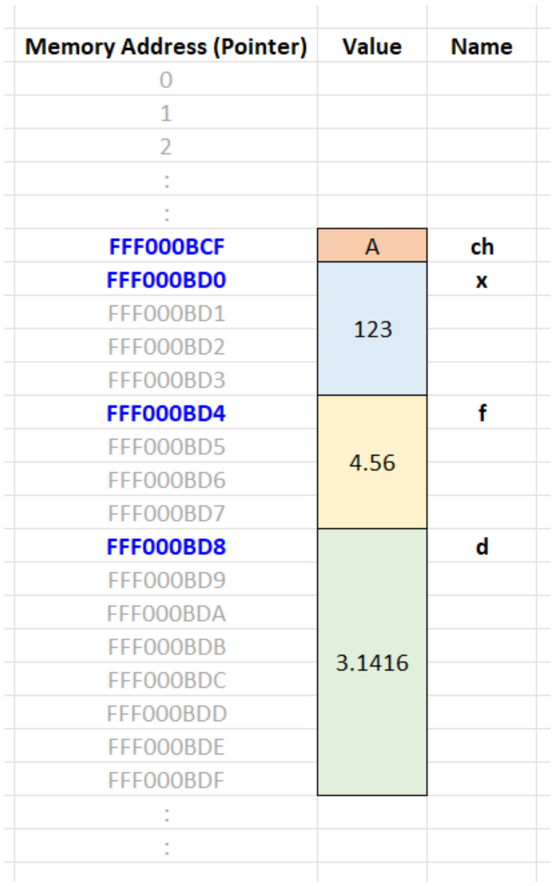
Figure 4.5: Completed source code, and the printf results

A visualization of the memory addresses, values and variables names are shown in Figure 4.6 (see next page). We'll refer to such a drawing as a **memory map**.

IMPORTANT:

- Recall that a computer's primary memory (the RAM) is made up of many bytes, for example 32GB.
- **Each byte in the memory has a unique address.** The first byte has a memory address of 0. The next byte has a memory address of 1, then 2, then 3, and so on…
- When a program is executed, each variable is allocated memory space for storing a value.
- The number of bytes allocated to a particular variable depends on its data type. A char variable requires 1 byte only, whereas the memory requirements of the int, float and double data types are platform dependent. Let's assume in this document that the int requires 4 bytes, float also requires 4 bytes and double requires 8 bytes.

As visualized in the memory map in Figure 4.6, character variable ch uses 1 byte which is at address FFF000BCF (we type the address in upper case for readability), integer variable x uses 4 bytes with memory addresses starting from FFF000BD0, float variable f also uses 4 bytes with addresses starting from FFF000BD4, and double variable d uses 8 bytes with addresses starting from FFF000BD8. **The address-of & operator will always give only the address of the first byte used by a variable.** These are the ones shown in blue color in the drawing.



Memory Address (Pointer)	Value	Name
0		
1		
2		
:		
:		
FFF000BCF	A	**ch**
FFF000BD0	123	**x**
FFF000BD1		
FFF000BD2		
FFF000BD3		
FFF000BD4	4.56	**f**
FFF000BD5		
FFF000BD6		
FFF000BD7		
FFF000BD8	3.1416	**d**
FFF000BD9		
FFF000BDA		
FFF000BDB		
FFF000BDC		
FFF000BDD		
FFF000BDE		
FFF000BDF		
:		
:		

Figure 4.6: Memory map showing the memory addresses (in hexadecimal), values, and names of variables

Learning Activity 4.3: Changing the value of a variable outside of the function where it was declared [incorrect way]

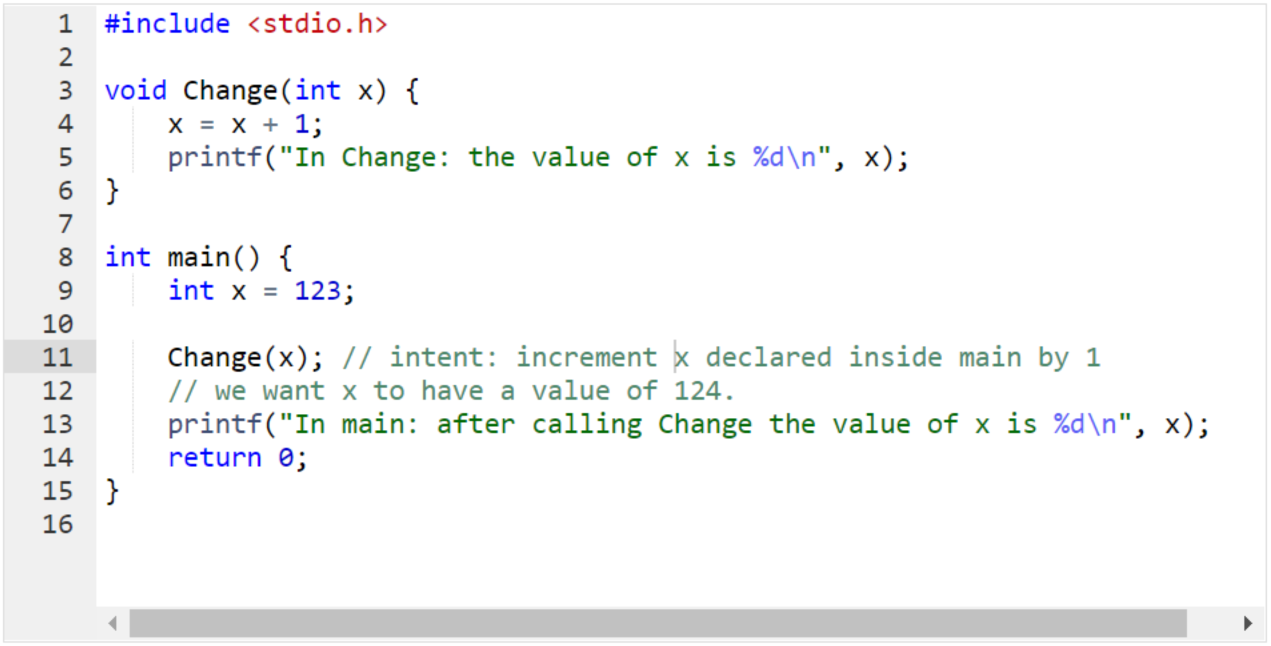
Why do we need to learn about memory addresses or pointers? We'll answer this question via a program that incorrectly tries to change the value of a variable outside of the function where it was declared.

Learning Outcome: The student can trace the control flow in function calls.

Instructions:

1. Encode the source code shown in Figure 4.7. Or if you want to omit the encoding step, you may simply click this link to open a browser window with a pre-encoded program.

   The intent of the program is to increment the value of x in main from 123 to 124. Instead of doing the increment inside main, we would like to do it in another function, specifically, inside Change.



```c
#include <stdio.h>

void Change(int x) {
    x = x + 1;
    printf("In Change: the value of x is %d\n", x);
}

int main() {
    int x = 123;

    Change(x); // intent: increment x declared inside main by 1
    // we want x to have a value of 124.
    printf("In main: after calling Change the value of x is %d\n", x);
    return 0;
}
```

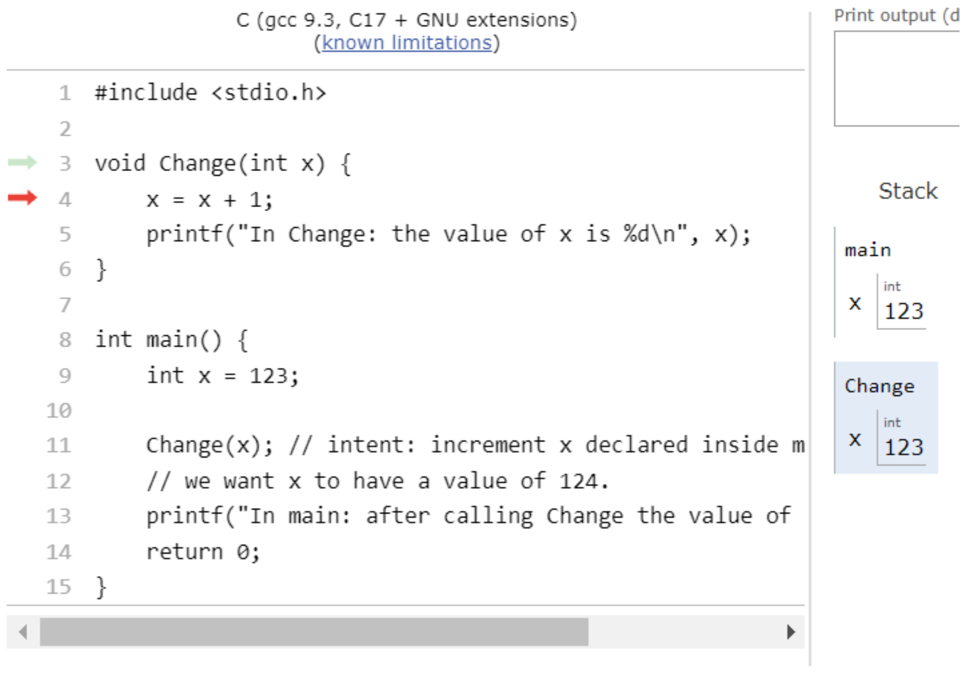
Figure 4.7: an incorrect attempt at changing the value of x defined in main

2. Trace and visualize the execution of the program by clicking on the "Next" button until the end of the program. Pay attention to what happens as you progress through each step.

   Did the program work based on the desired intent?

In Figure 4.8, notice that the visualization for the two variables which happen to have the same name x in the program. One of them is the x that was defined inside main in Line 9, and the other one is the formal parameter x specified in function Change in Line 3. We already learned that **in parameter passing, what is passed is the value of the parameter**. Thus, in Line 11, what was passed was the value of x which is 123, which was then copied as the initial value of formal parameter x specified in Line 3.

The effect of the increment in Line 4 was made in formal parameter x in Change, but this did not, in any way, produced the intended effect on variable x in main. This is the cause of the logical error.



```
 1  #include <stdio.h>
 2
 3  void Change(int x) {
 4      x = x + 1;
 5      printf("In Change: the value of x is %d\n", x);
 6  }
 7
 8  int main() {
 9      int x = 123;
10
11      Change(x); // intent: increment x declared inside m
12      // we want x to have a value of 124.
13      printf("In main: after calling Change the value of
14      return 0;
15  }
```

Figure 4.8: Variable x in main is different from the variable x in Change.

Learning Activity 4.4: Changing the value of a variable outside of the function where it was declared [CORRECT way]

The value of a variable cannot be changed outside of the function where it was declared by DIRECT memory access. The only way to do it is via INDIRECT memory access. Indirect memory access requires us to be knowledgeable in using the **indirection operator** denoted by the asterisk symbol *.

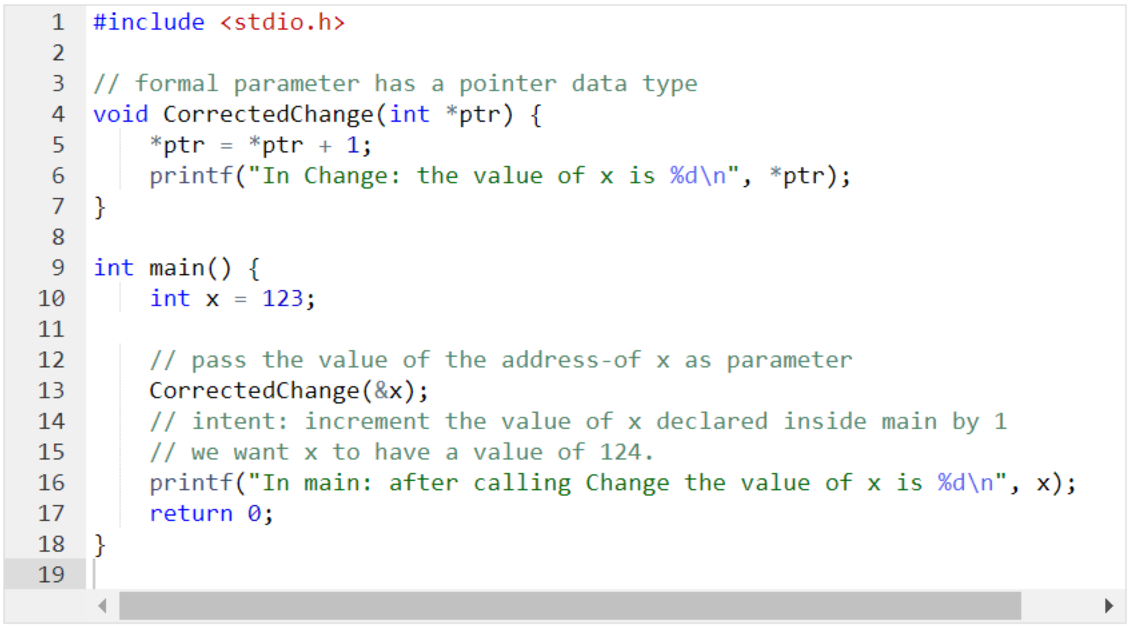
Learning Outcome: The student can trace a code containing pointers and indirection operations.

Instructions:

1. Encode the source code shown in Figure 4.9. Or if you want to omit the encoding step, you may simply click this link to open a browser window with a pre-encoded program.

   Again, the intent of the program is to increment the value of x in main from 123 to 124.

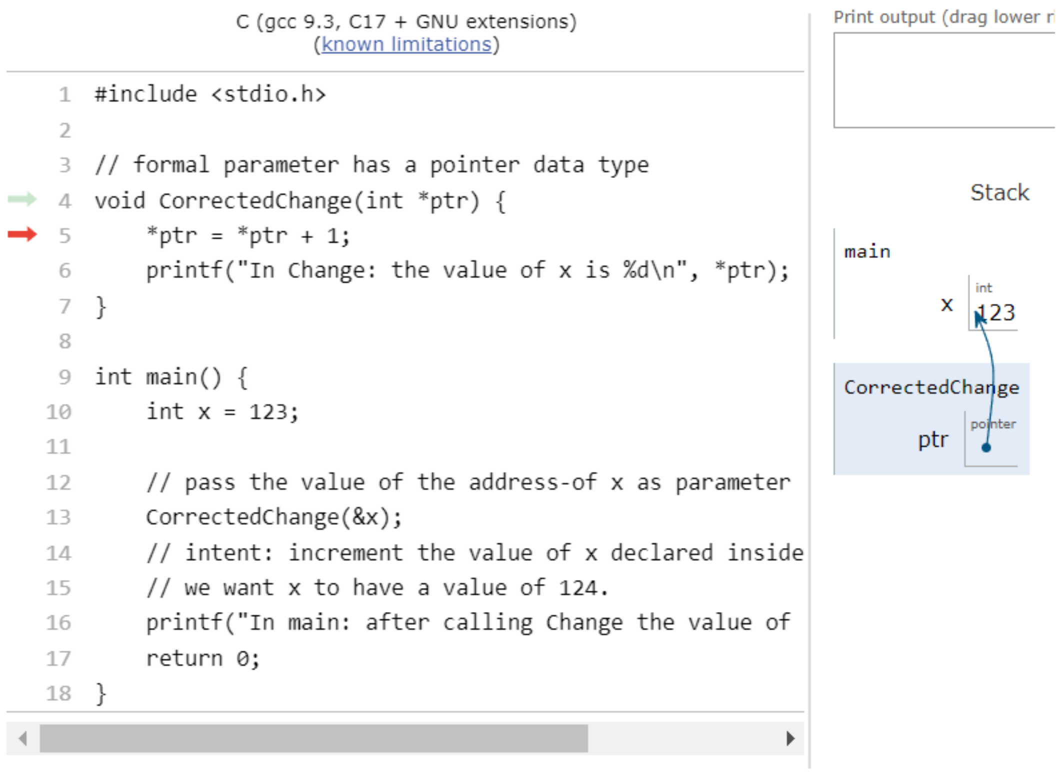
   Three things had to be done to correctly effect the desired change, specifically:
   - In Line 13: we pass the value of the address-of x, i.e., &x as parameter.
   - In Line 4: we specify the data type of the formal to a pointer data type, specifically **int *** (note: int * is read as "integer pointer"). Although not necessary, we used **ptr** as formal parameter name for reason of readability.
   - In Line 5: we access the memory space allocated to x INDIRECTLY via the indirection operator denoted by the asterisk symbol *.



```
 1  #include <stdio.h>
 2
 3  // formal parameter has a pointer data type
 4  void CorrectedChange(int *ptr) {
 5      *ptr = *ptr + 1;
 6      printf("In Change: the value of x is %d\n", *ptr);
 7  }
 8
 9  int main() {
10      int x = 123;
11
12      // pass the value of the address-of x as parameter
13      CorrectedChange(&x);
14      // intent: increment the value of x declared inside main by 1
15      // we want x to have a value of 124.
16      printf("In main: after calling Change the value of x is %d\n", x);
17      return 0;
18  }
19
```

Figure 4.9: correct way of changing the value of x outside of main where it was defined

2. Trace and visualize the execution of the program by clicking on the "Next" button until the end of the program. Pay attention to what happens as you progress through each step.


C (gcc 9.3, C17 + GNU extensions)
(known limitations)

```
#include <stdio.h>

// formal parameter has a pointer data type
void CorrectedChange(int *ptr) {
    *ptr = *ptr + 1;
    printf("In Change: the value of x is %d\n", *ptr);
}

int main() {
    int x = 123;

    // pass the value of the address-of x as parameter
    CorrectedChange(&x);
    // intent: increment the value of x declared inside
    // we want x to have a value of 124.
    printf("In main: after calling Change the value of
    return 0;
}
```

Figure 4.10: visualization of ptr (the pointer variable) as a directed arrow

Figure 4.10 shows a screenshot of step 5. Notice the visualization for the variables x and ptr on the right half of the screenshot. Just like before, x contains a value of 123. What's new here is the visualization for the pointer variable ptr. Notice that the word "pointer" is shown in the visualization, and instead of a value, a directed arrow symbol in blue color is drawn such that it is "pointing" to the memory space associated with variable x. The directed arrow is C Tutor's way of visualizing the phrase "ptr is pointing to x" which means that the value of variable ptr is the memory address of x. **In other words, ptr == &x.**

The statement ***ptr = *ptr + 1;** in Line 5 is the INDIRECT way to access, i.e., increment, the value stored in the memory space associated with variable x in main.

The program does the following internally (although we programmers do not see it):

Given the expression

***ptr = *ptr + 1**

Substitute the value of **&x** to **ptr** since **ptr == &x**, i.e.,

***&x = *&x + 1**

The two consecutive operations ***&** (INDIRECTION operator followed by address-of operator) will cancel each other out, thus resulting into the simpler expression:

x = x + 1

This proves that the two expressions **x = x + 1** (DIRECT memory access) and ***ptr = *ptr + 1** (INDIRECT memory access) produce the same effect.

IMPORTANT: Once we have set the value of **ptr = &x** (which was achieved in the program above via parameter passing), any occurrence of **x** can be replaced with the indirection operation ***ptr**. That is, **x** is the DIRECT access, while ***ptr** is the INDIRECT access.

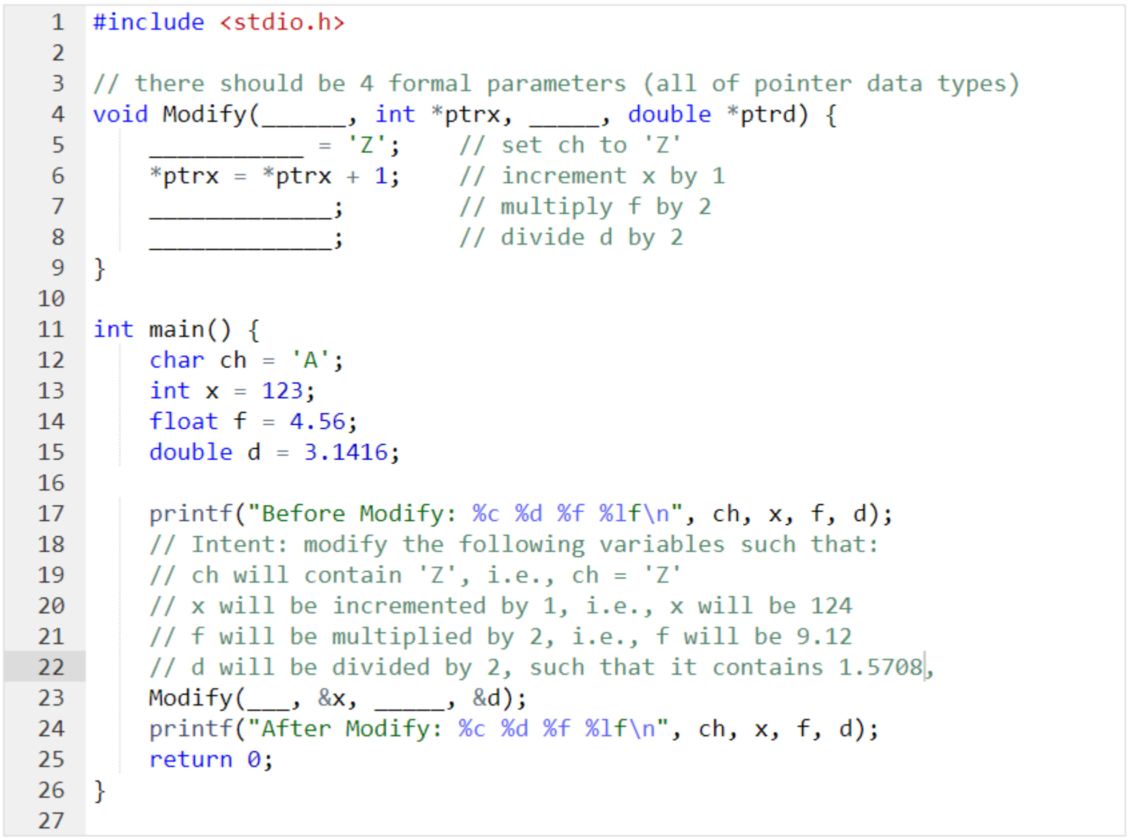
NOTE: the exposition above was probably more than what was covered in your own CCPROG1 class discussions. Please consider it as an additional or supporting explanation.

Learning Activity 4.5: Indirect memory access of several variables

Learning Outcome: The student can write a program containing indirect memory access operations

Instructions:

1. Encode and complete the missing parts of the source code shown in Figure 4.11. Or if you want to omit the encoding step, you may simply click this link to open a browser window with a pre-encoded program.



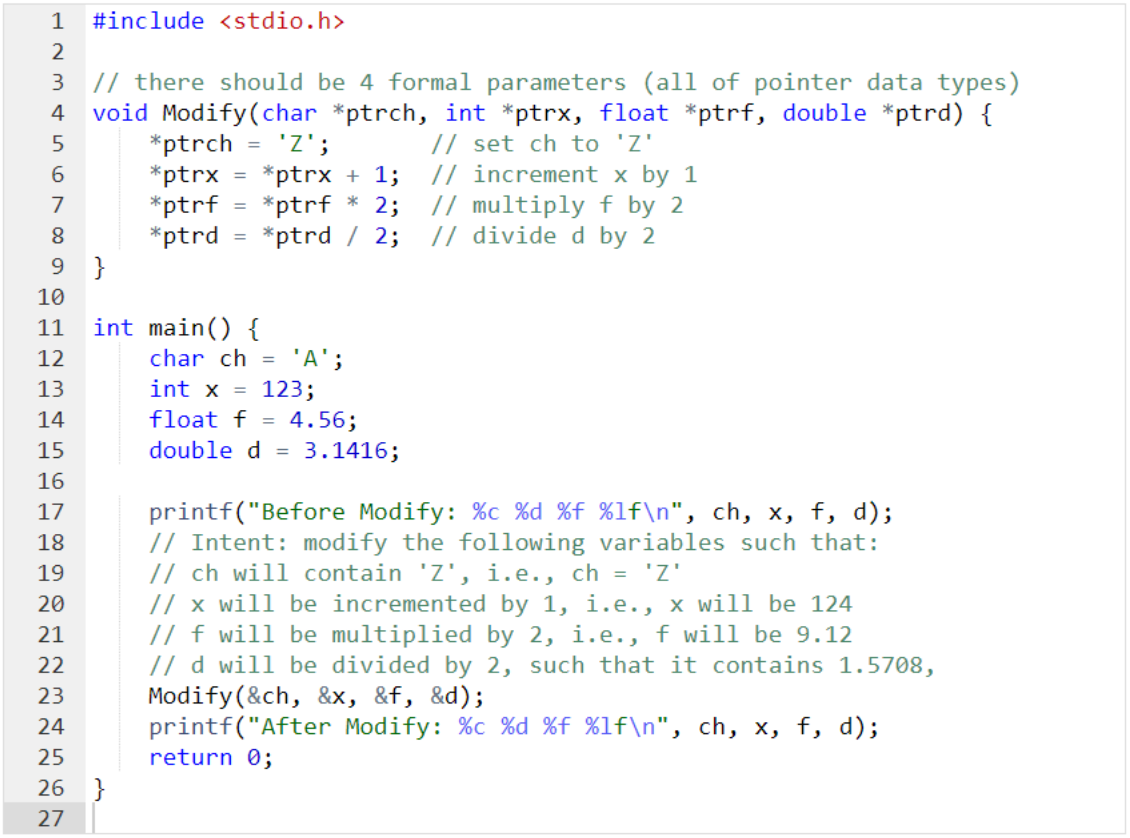
```c
#include <stdio.h>

// there should be 4 formal parameters (all of pointer data types)
void Modify(______, int *ptrx, _____, double *ptrd) {
    ___________ = 'Z';    // set ch to 'Z'
    *ptrx = *ptrx + 1;    // increment x by 1
    _____________;        // multiply f by 2
    _____________;        // divide d by 2
}

int main() {
    char ch = 'A';
    int x = 123;
    float f = 4.56;
    double d = 3.1416;

    printf("Before Modify: %c %d %f %lf\n", ch, x, f, d);
    // Intent: modify the following variables such that:
    // ch will contain 'Z', i.e., ch = 'Z'
    // x will be incremented by 1, i.e., x will be 124
    // f will be multiplied by 2, i.e., f will be 9.12
    // d will be divided by 2, such that it contains 1.5708,
    Modify(___, &x, _____, &d);
    printf("After Modify: %c %d %f %lf\n", ch, x, f, d);
    return 0;
}

```

Figure 4.11: Fill-up the missing parts of the program

2. Trace and visualize the execution of the program by clicking on the "Next" button until the end of the program. Pay attention to what happens as you progress through each step.

   Please DO NOT proceed to the next page without trying this activity. The answer is found on the next page.

The complete code is shown below for your reference.   If you want to omit the encoding step, you may simply click this link to open a browser window with a pre-encoded program.



```c
#include <stdio.h>

// there should be 4 formal parameters (all of pointer data types)
void Modify(char *ptrch, int *ptrx, float *ptrf, double *ptrd) {
    *ptrch = 'Z';       // set ch to 'Z'
    *ptrx = *ptrx + 1;  // increment x by 1
    *ptrf = *ptrf * 2;  // multiply f by 2
    *ptrd = *ptrd / 2;  // divide d by 2
}

int main() {
    char ch = 'A';
    int x = 123;
    float f = 4.56;
    double d = 3.1416;

    printf("Before Modify: %c %d %f %lf\n", ch, x, f, d);
    // Intent: modify the following variables such that:
    // ch will contain 'Z', i.e., ch = 'Z'
    // x will be incremented by 1, i.e., x will be 124
    // f will be multiplied by 2, i.e., f will be 9.12
    // d will be divided by 2, such that it contains 1.5708,
    Modify(&ch, &x, &f, &d);
    printf("After Modify: %c %d %f %lf\n", ch, x, f, d);
    return 0;
}

```

--- End of Module 4.   Up Next: Tracing and visualizing loops. --

The answers to Check-up Quiz #3 are shown in the footnote below[1].

[1] Q1: What is the symbol for the **address-of operator**?
A1: &
Q2: What is the symbol for the **indirection (or dereference) operator**?
A2: *
Q3: Given the source code below, what is the value of **x** that will be displayed by the printf() statement in Line 13?
A3: 123
Q4: Given the source code below, what is the value of **x** that will be displayed by the printf() statement in Line 16?
A4: 124
Q5: Given the source code below, what is the data type of variable **x** defined in Line 10?
A5: int
Q6: Given the source code below, what is the data type of the expression **&x** in Line 13?
A6: int *
Q7: Given the source code below, what is the data type of the formal parameter **ptr** in Line 4?
A7: int *
Q8: Given the source code below, what is the data type of the expression ***ptr** in Line 5?
A8: int