Introduction to Scientific and Engineering Computation (BIL 104E)

Lecture 8

Pointers

Pointers

Pointers represent one of the most important and powerful features in C.

Pointers can be handled with topics are given:

- **Pointer variables**
- **Memory addresses**
- **The concept of indirection**
- **Declaring a pointer**
- **The address-of operator**
- **The dereference operator**

Pointers: What Is a Pointer?

Until now, you have only learnt variables directly, by assigning values to them.

After then, new concept, known as **indirection** will be introduced by using pointer concept.

Definition:
Instead of assigning values directly to variables, a variable can be indirectly manipulated by creating a variable called **a pointer**, which contains the **memory address** of another variable.

Pointers: Why Is the Pointer So Important?

- Using the memory address of your data is often the quickest and simplest way to access it.
- There are many things that are difficult without pointers:
  - dynamically allocating memory
  - passing large data structures between functions
  - talking to your computer's hardware.

You should know two things about pointer :

1-) A pointer is a variable, so you can assign different values to a pointer variable.

2-) The value contained by a pointer must be an address that indicates the location of another variable in the memory.

That's why a pointer is also called an address variable.

Pointers: Address (Left Value) Versus Content (RightValue)

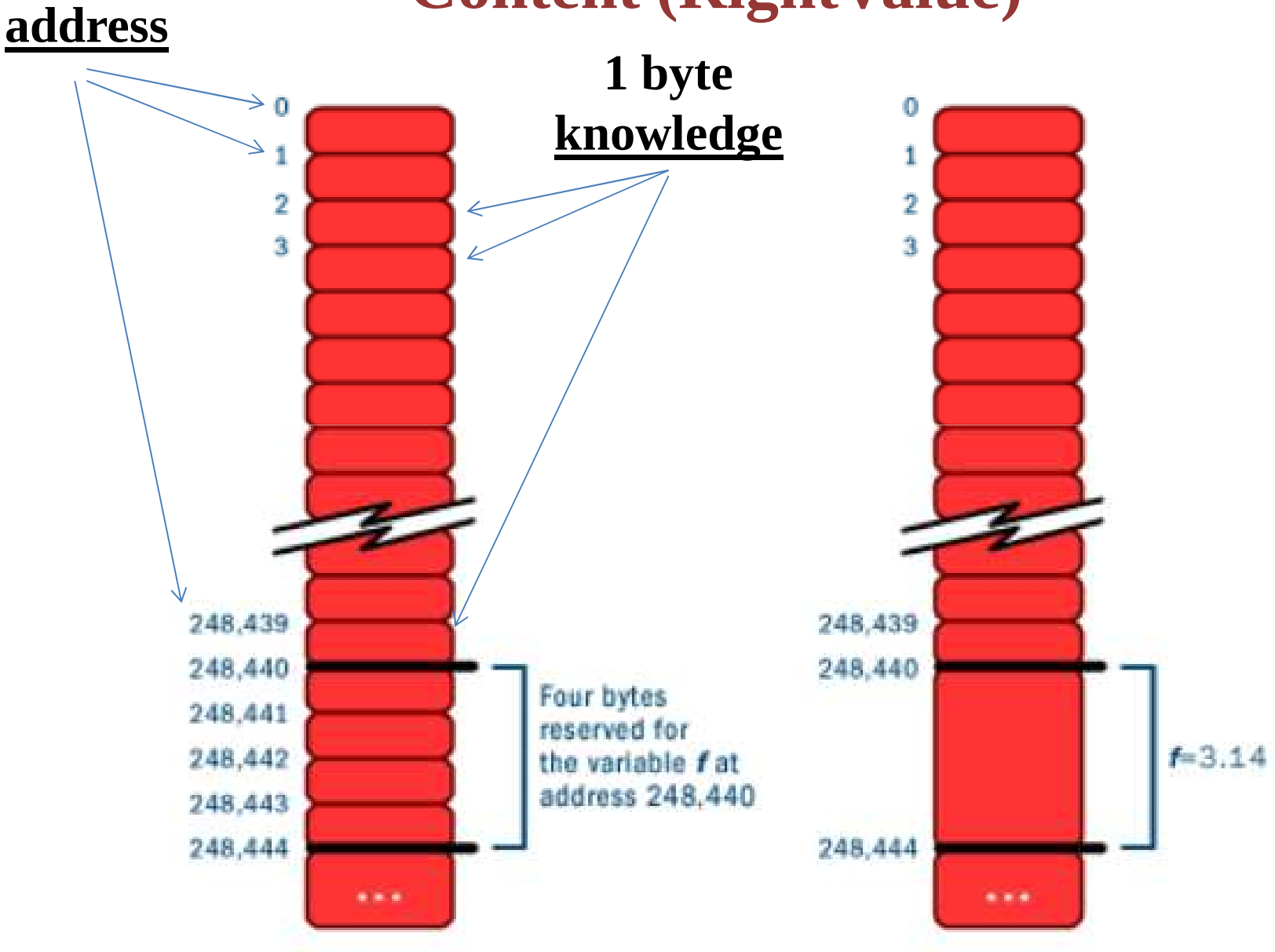
The memory inside your computer is used to hold the binary code of your program.

Each memory location must have a unique address so that the computer can read from or write to the memory location without any confusion.
For instance : Each house in a city must have a unique address.

When a variable is declared, a piece of unused memory will be reserved for the variable, and the unique address to the memory will be associated with the name of the variable.

The address associated with the variable name is usually called the **left value** of the variable.

Pointers: Address (Left Value) Versus Content (RightValue)



Pointers: Address (Left Value) Versus Content (RightValue)

When the variable is assigned a value, the value is stored into the reserved memory location as the content. The content is also called the **right value** of the variable.

For instance:

After the integer variable **x** is declared and assigned to a value

```
int x;
x = 7;
```

the variable **x** now has two values:

Left value: 1000

Right value: 7

the left value, **1000**, is the address of the memory location reserved for **x**. The right value, **7**, is the content stored in the memory location.

Pointers: Address (Left Value) Versus Content (RightValue)

You can imagine that the variable x is the mailbox in front of your house, which has the address 1000. The right value, 7, can be thought of as a letter delivered to the mailbox.

When a C program is being compiled and a value is being assigned to a variable, the C compiler has to check the left value of the variable.

If the compiler cannot find the left value, it will issue an error message saying that the variable is undefined in your program.

That’s why, in C, you have to declare a variable before you can use it.

Pointers: The Address-of Operator (&)

If you want to know the left value of a variable or memory address, **the address-of operator** , **&** , can be used to evaluates to the address (the left value) of a variable.

The following code, for example,

```
long int x;
long int *y;
y = &x;
```

assigns the address of the long integer variable x to a pointer variable, y.

Pointers: The Address-of Operator (&)

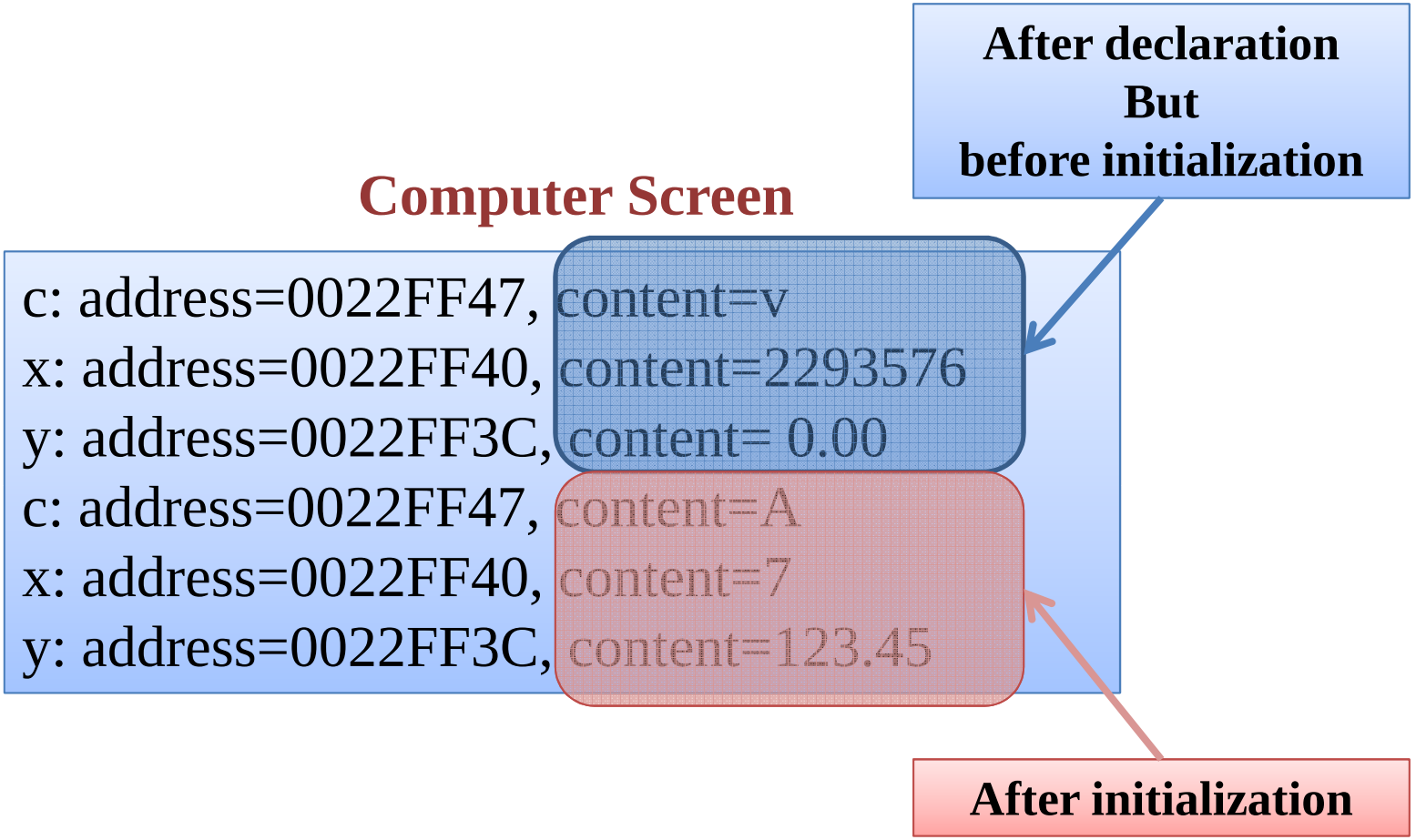
Obtaining the Left Values of Variables

```
1:  /* 11L01.c: Obtaining addresses */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char  c;
7:     int   x;
8:     float y;
9:
10:    printf("c: address=%p, content=%c\n", &c, c);
11:    printf("x: address=%p, content=%d\n", &x, x);
12:    printf("y: address=%p, content=%5.2f\n", &y, y);
13:    c = 'A';
14:    x = 7;
15:    y = 123.45;
16:    printf("c: address=%p, content=%c\n", &c, c);
17:    printf("x: address=%p, content=%d\n", &x, x);
18:    printf("y: address=%p, content=%5.2f\n", &y, y);
19:    return 0;
20: }
```

If you do not initialize variable with a value, compiler will allocate a memory part for variable after its declaration and you can see old value located in this area.

After initialization, you can see the new initial value located in the same area for the variable.

Pointers: The Address-of Operator (&)



Pointers: Declaring Pointers

The general form of a pointer declaration is

data-type *pointer-name;

Here **data-type** specifies the type of data to which the pointer points. **pointer-name** is the name of the pointer variable, which can be any valid variable name in C.

In front of the pointer name is an **asterisk ***, which indicates that the variable is a pointer.
When the compiler sees the **asterisk** in the declaration, it considers that the variable can be used as a pointer.

The following shows different types of pointers:

```
char  *ptr_c; /* declare a pointer to a character */
int   *ptr_int; /* declare a pointer to an integer */
float *ptr_flt; /* declare a pointer to a floating-point */
```

Pointers: Declaring Pointers

Declaring and Assigning Values to Pointers

```
1:  /* 11L02.c: Declaring and assign values to pointers */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:      char  c, *ptr_c;
7:      int   x, *ptr_x;
8:      float y, *ptr_y;
9:
10:     c = 'A';
11:     x = 7;
12:     y = 123.45;
13:     printf("c: address=%p, content=%c\n", &c, c);
14:     printf("x: address=%p, content=%d\n", &x, x);
15:     printf("y: address=%p, content=%5.2f\n", &y, y);
16:     ptr_c = &c;
17:        printf("ptr_c: address=%p, content=%p\n", &ptr_c, ptr_c);
18:        printf("*ptr_c => %c\n", *ptr_c);
19:     ptr_x = &x;
20:        printf("ptr_x: address=%p, content=%p\n", &ptr_x, ptr_x);
21:        printf("*ptr_x => %d\n", *ptr_x);
22:     ptr_y = &y;
23:        printf("ptr_y: address=%p, content=%p\n", &ptr_y, ptr_y);
24:        printf("*ptr_y => %5.2f\n", *ptr_y);
25:     return 0;
26: }
```

Pointer declaration

Address assignment to the pointer

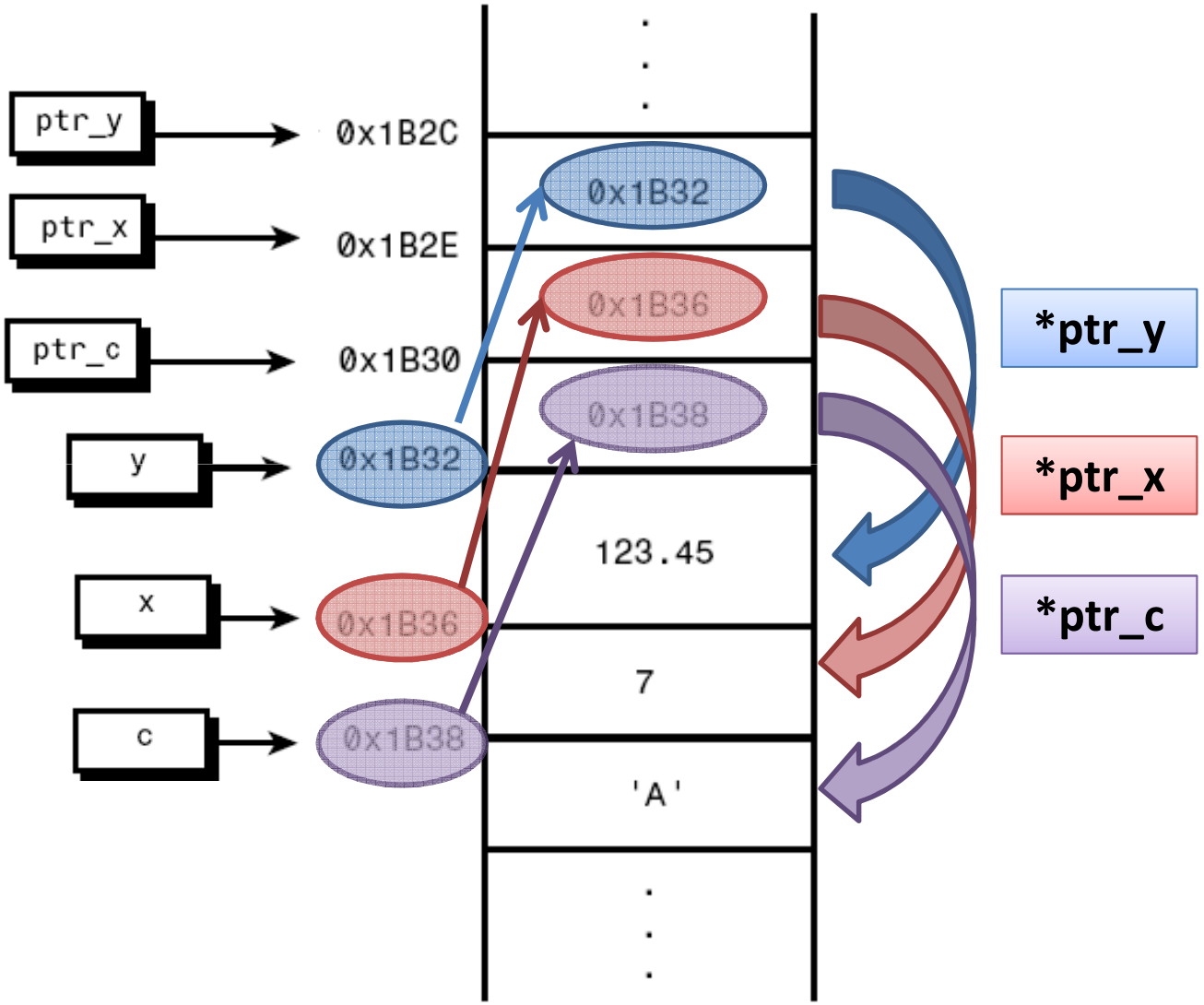
Reaching variable pointed by pointer

Pointers: Declaring Pointers

Computer Screen

```
c: address=0x1B38, content=A
x: address=0x1B36, content=7
y: address=0x1B32, content=123.45
ptr_c: address=0x1B30, content=0x1B38
*ptr_c => A
ptr_x: address=0x1B2E, content=0x1B36
*ptr_x => 7
ptr_y: address=0x1B2C, content=0x1B32
*ptr_y => 123.45
```

Pointers: Declaring Pointers



Pointers: The Dereference Operator (*)

The asterisk is called the **dereference operator** when it is used as a unary operator. (Sometimes, it's also called the indirection operator.)

The value of a variable can be referenced by the combination of the **asterisk *** and the address of the variable as the operand.

This combination is also used for the declaration of a pointer.

For instance:

```
int  x;   //decleration of integer variable x
int  *ptr_y;  //decleration of pointer  ptr_y

x = 5;              //  5 is assigned to variable x
ptr_y = &x;  //  address of varible x is assigned to ptr_y
*ptr_y = 6;   // using ptr_y,  6 is assigned to varible x
```

Pointers: The difference between dereference and multiplication operators

Don’t confuse the **dereference operator** with **the multiplication operator**, although they share the same symbol, *.

The dereference operator is a unary operator, which takes only one operand. The operand contains the address of a variable.

On the other hand, the multiplication operator is a binary operator that requires two operands to perform the operation of multiplication.

For instance:

```
*ptr_y      // ptr_y should be a pointer and can contain only
               address of a variable (left value).

x * y       // x and y should be right value for multiplication.
```

Pointers: Null Pointers

A null pointer can never point to valid data.

If zero is assigned to a pointer, it is called as null pointer

For instance:

```
char   *ptr_c;
int   *ptr_int;

ptr_c = ptr_int = 0;
```

ptr_c and **ptr_int** become **null pointers** after 0 is assigned to them.

Pointers: Updating Variables via Pointers

If you take the address of a variable, you can change the value of the variable by using **dereference operator ***.

Changing Variable Values Via Pointers

```
1:  /* 11L03.c: Changing values via pointers */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char  c,  *ptr_c;
7:
8:     c = 'A';
9:     printf("c: address=%p, content=%c\n", &c, c);
10:    ptr_c = &c;
11:       printf("ptr_c: address=%p, content=%p\n", &ptr_c, ptr_c);
12:       printf("*ptr_c => %c\n", *ptr_c);
13:    *ptr_c = 'B';
14:       printf("ptr_c: address=%p, content=%p\n", &ptr_c, ptr_c);
15:       printf("*ptr_c => %c\n", *ptr_c);
16:    printf("c: address=%p, content=%c\n", &c, c);
17:    return 0;
18: }
```

Pointers: Updating Variables via Pointers

Computer Screen

```
c: address=0x1828, content=A
ptr_c: address=0x1826, content=0x1828
*ptr_c => A
ptr_c: address=0x1826, content=0x1828
*ptr_c => B
c: address=0x1828, content=B
```

Pointers: Pointing to the Same Memory Location

A memory location can be pointed to by more than one pointer.

For instance:

given that c = ‘A’ and that ptr_c1 and ptr_c2 are two character pointer variables,

ptr_c1 = &c and ptr_c2 = &c set the two pointer variables to point to the same location in the memory.

Pointers: Pointing to the Same Memory Location

Pointing to the Same Memory Location with More Than One Pointer

```
1:  /* 11L04.c: Pointing to the same thing */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int x;
7:     int *ptr_1, *ptr_2, *ptr_3;
8:
9:     x = 1234;
10:    printf("x: address=%p, content=%d\n", &x, x);
11:    ptr_1 = &x;
12:    printf("ptr_1: address=%p, content=%p\n", &ptr_1, ptr_1);
13:       printf("*ptr_1 => %d\n", *ptr_1);
14:    ptr_2 = &x;
15:    printf("ptr_2: address=%p, content=%p\n", &ptr_2, ptr_2);
16:       printf("*ptr_2 => %d\n", *ptr_2);
17:    ptr_3 = ptr_1;
18:    printf("ptr_3: address=%p, content=%p\n", &ptr_3, ptr_3);
19:       printf("*ptr_3 => %d\n", *ptr_3);
20:    return 0;
21: }
```

Pointers: Pointing to the Same Memory Location

Computer Screen

```
x: address=0x1838, content=1234
ptr_1: address=0x1834, content=0x1838
*ptr_1 => 1234
ptr_2: address=0x1836, content=0x1838
*ptr_2 => 1234
ptr_3: address=0x1832, content=0x1838
*ptr_3 => 1234
```