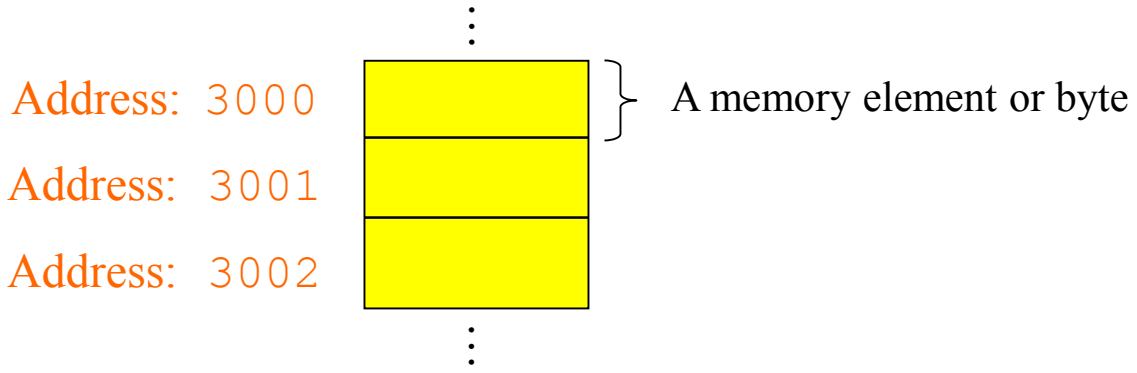
Lecture 12:

Pointers and Addresses

Prof. Shervin Shirmohammadi
University of Ottawa

Storage of Variables in Memory

- Along with the executable image of a C program, all variables must be stored in main memory (RAM).
- Main memory is organized as a large collection of individual memory elements.
  - An individual memory element (word or byte) is the smallest accessible memory unit.
- Each memory element in a computer has a unique address that specifies its position or location in memory.
  - The address of a memory element is a positive integer.
  - Contiguous memory elements have contiguous addresses:



Concept of allocated memory in C

- The number of memory elements required to store variables depends on:
  - the type of the variable to be stored; ie: `int`, `float`…, and
  - the computer system on which the program is compiled.
- The operator `sizeof` can be used to determine the amount of memory elements required to store a particular object.
  - The operator `sizeof` requires one argument which can be a type or variable name enclosed within parentheses.
  - The operator `sizeof` yields the number of memory elements required to store the argument.
  - If the argument is the name of an array then the total number of bytes required to store the array is obtained.

Example

- Write a program to print the size of simple variables we have often used in the course; i.e., *int*, *float*, and *double*.

Concept of allocated memory in C

- On my PC, the following numbers are obtained from the operator `sizeof`:

E.g.: `int a, b[3];`

`float x;`

`sizeof(int);` ⟹ yields 2

`sizeof(a);` ⟹ yields 2

`sizeof(b);` ⟹ yields 6

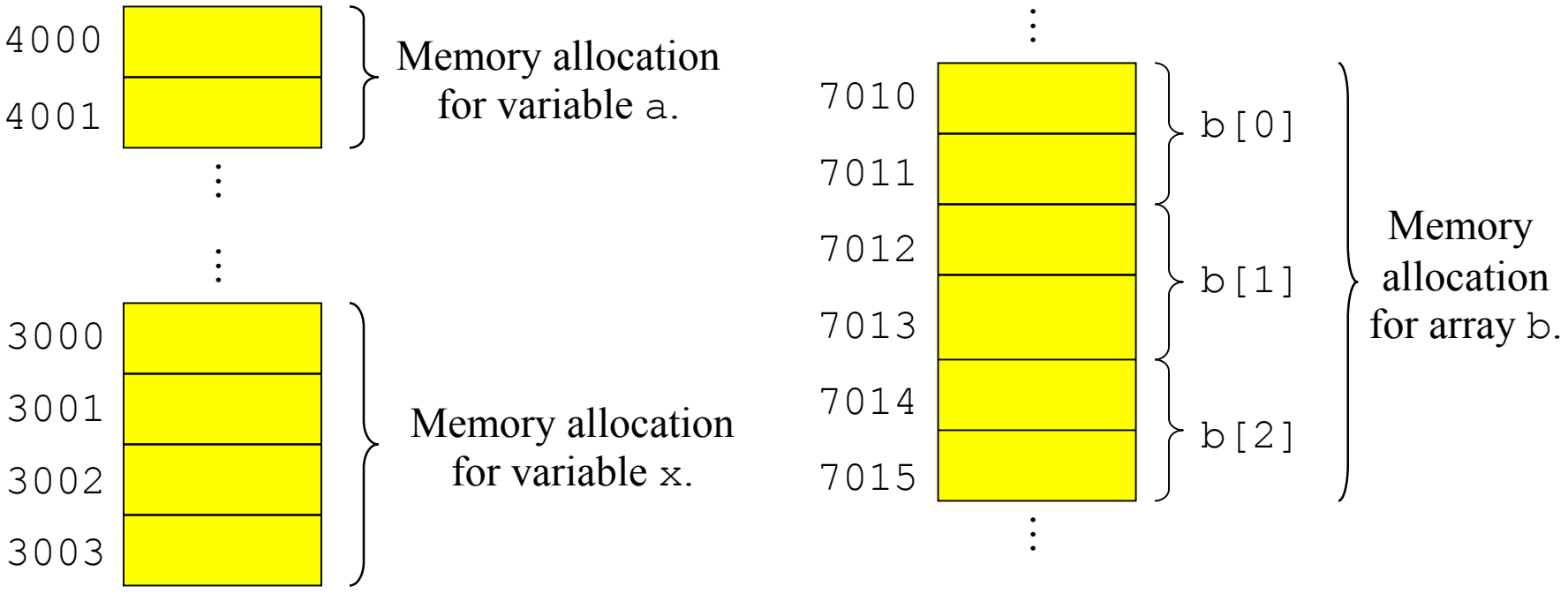
`sizeof(float);` ⟹ yields 4

`sizeof(x);` ⟹ yields 4

- During the compilation of a C program, the memory required to store all variables is known. At compilation time, the memory for global variables is reserved and their addresses is determined. When the program is loaded into memory prior to execution, the exact location of the global variables is known.

- E.g.: Our declarations of the following global variables could result in the following allocation of memory upon loading:

```
int a, b[3];
    ⋮
float x;
```



- For local variables (including parameters) the allocation of memory is done when the function is executed

Remarks

- Contiguous memory elements are always used to store a single variable.
- The elements of an array are guaranteed to be stored in contiguous memory locations.
- Variables of the same type defined in the same declaration are **not** necessarily stored in contiguous memory locations.
- The location of local variables (parameters) in memory **cannot** be known ahead of time since they are determined when the function is executed.

Concept of Addresses in C

- When a program and its variables are loaded into memory prior to execution, the exact location or address of all variables in memory becomes known.
- The address of a variable in main memory is the address of the first byte of the memory used to store the variable.
- The address of a variable can be obtained in C using the address operator which is the **ampersand**: `&`
  - The address operator is unary and operates on the argument directly to its right.
  - The argument of the address operator must be a variable
    - The argument cannot be a symbolic constant or an expression.
  - The address operator returns the address of the first memory element used to store the argument:

    E.g.:
    ```
    int integer;
    &integer;
    ```
    - Recall that this is a positive integer which can change from one execution to another and is known only after loading.
- The pseudo-code expression ***The address of integer*** in C becomes `&integer`

Addresses, variables, and contents

- Addresses, variable names and contents: an analogy with a postal box.

Postal box number	Individual	Contents
490	**John Doe**	**Newspaper**
Memory address	Variable name	Contents
`6572`	`ctr`	`10`

  - The address of a memory element is similar to the number of the postal box.
  - The memory element in location `6572` is assigned to the variable `ctr` just like postal box 490 is assigned to John Doe.
  - The variable `ctr` contains the number `10` and the postal box contains a newspaper.
  - This analogy is not completely correct since:
    - a postal box may contain many items while a memory element can only contain one quantity or part of one quantity,
    - it is possible for two individuals to have the same name but two variables in a program must have different names,
    - many individuals may share the same postal box but variables in a program must have distinct memory assignments.
    - The memory address of a local variable changes from one execution of the function to the next execution while the postal box number remains contant (resembles more the address of the global variable).
    - During the execution of a program, the computer does not use variable names, just addresses.

Concept: the pointer – a special variable

- The pointer is ***a variable*** that contains an address
- In pseudo-code, lets add ***Ptr*** to the name of a pointer variable
  - Example: ***integerPtr***
  - And also:
    - ***Assign the address of integer to integerPtr***

Pointers in C: Declaration and Initialization

- A pointer is a variable that will contain an address.
  - A *variable* contains a *value* and a **pointer** contains the **address of a variable**.
- The declaration of a pointer must specify the type of the variable to which it will point as well as the name of the pointer.
  - E.g.:

```
int integer, *integerPtr;
float real, *realPtr;
```

  - The above declarations specify that:

    `integer` is a variable of type `int`,

    `integerPtr` is a pointer that points to a variable of type `int`,

    `real` is a variable of type `float`,

    `realPtr` is a pointer that points to a variable of type `float`.
  - The asterisk ***** in a declaration signifies that the name that follows is a pointer.
  - The letters `Ptr` are often used in the name of a pointer to render the name more descriptive
    - It signals to the reader that the name is a pointer variable.

- A pointer to a variable of one type cannot be used to point to a variable of another type.
  - E.g.: A pointer to a variable of type `int` cannot be used to point to a variable of type `float`.
- A pointer points to nowhere in particular until it is **initialized**.
  - In our previous examples, `integerPtr` and `realPtr` do not yet point to the variables `integer` and `real` even if the variable and its pointer appear in the same declaration.
- To initialize a pointer, we must assign it an address, and the address is that of the variable to which we want it to point.
  - E.g.:

```
int integer, *integerPtr;
float real, *realPtr;

integer = 7;
real = 10.0;

integerPtr = &integer;
realPtr = &real;
```

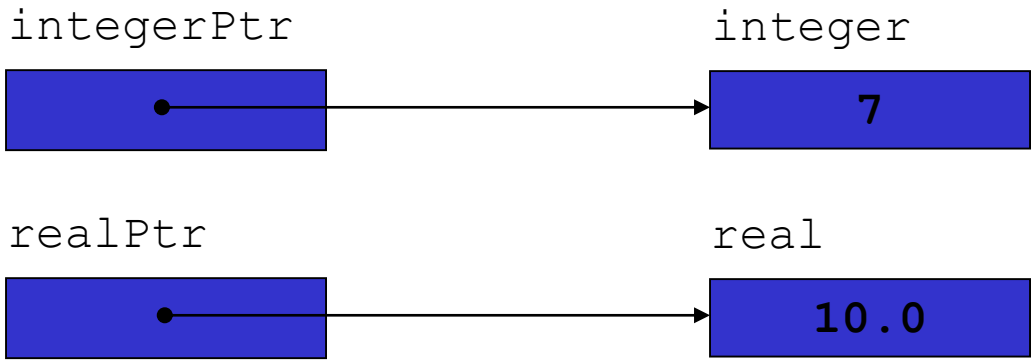
Declarations.

Variable initializations.

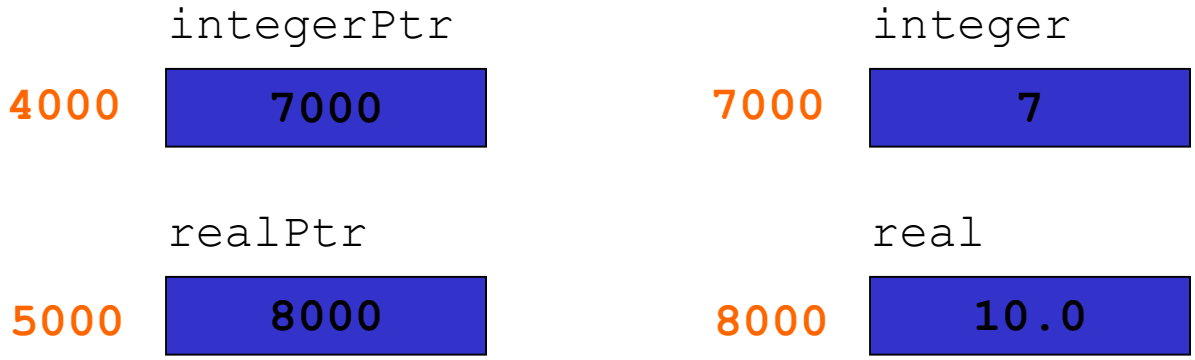
Pointer initializations.

- A variable can have more than one pointer pointing to it.

- After the previous initializations, we can visualize the memory contents as shown below:



- Assuming addresses, the organization in memory and content of our variables and pointers would be more like:



  - Thus if `integer` and `real` were located at addresses `7000` and `8000`, respectively, then our pointers `integerPtr` and `realPtr` would contain these addresses after initialization.
  - Note that pointers also have addresses since they are themselves variables.
  - **Remember: it is only after loading that actual addresses become known.**

Indirection or dereferencing

- The pointer can be used to read the contents of the referenced memory by the pointer (i.e. the content of the referenced variable)
- This is called indirection or dereferencing
- In pseudo-code, express indirection with
  - ***The content pointed by*** *namePointer*
  - *Examples*

    Assign 7 to integer

    Assign the address of integer to integerPtr

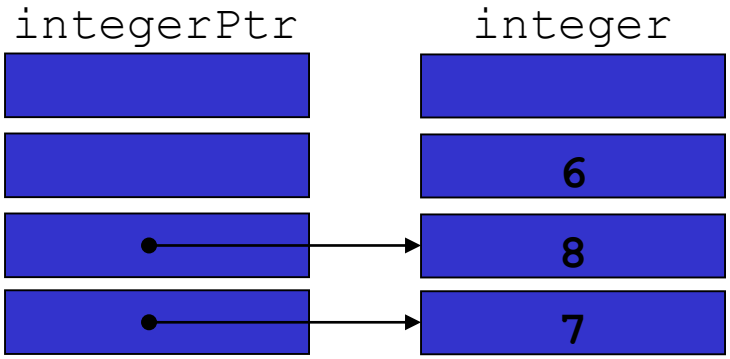
    Print the content pointed by integerPtr

    Assign 8 the content pointed by integerPtr

Pointers: Indirection in C

- An initialized variable contains a value.
  - We can access this value by specifying the name of the variable.
- An initialized pointer contains the address of a variable which contains a value.
  - The value in the variable can be accessed through the variable's pointer using the indirection or dereferencing operator.
- The indirection or dereferencing operator is the asterisk `*`. It is a unary operator which operates directly on the argument to its right.
  - E.g.:

```
int integer, *integerPtr;
integer = 8;
integerPtr = &integer;
*integerPtr = 7;
```



  - The first line declares the variable `integer` and the pointer `integerPtr`,
  - the second line initializes the variable `integer`,
  - the third line initializes the pointer `integerPtr`,
  - the fourth line ***Assigns 7 to the content pointed by integerPtr***; i.e.: in `integer`

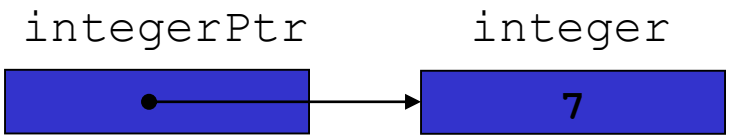
Example

- What will be the output of the following *printf* statements? Try the code in a program.

```
int X, *PtrToX;
X = 5;
PtrToX = &X;

printf("X is %d", X);
printf("\nPtrToX is %d", PtrToX);
printf("\nThe integer pointed to by PtrToX is %d",
   *PtrToX);
printf("\nThe address of PtrToX itself is %d",
   &PtrToX);
```

Pointers: Indirection in C



- "Memory aids" for dereferencing a pointer:
  - In pseudo-code express assignments like:

    ```
    *integerPtr = 7;
    ```

    as: *Assign 7 to the content pointed by integerPtr*
  - Or in the case of using a value reference by a pointer:

    ```
    printf("The content of integer is %d",
      *integerPtr);
    ```

    Write in pseudo-code:

    Print "The content of integer is ", the content pointed by integerPtr

- The addressing and indirection operators are complementary.
  - E.g.: assume that `integer` and `integerPtr` have been declared and initialized, and are organized as follows in memory:

  - Now consider the following operations and results:
    - `*&integerPtr` ➔ `&integerPtr` yields `4000` and `*&integerPtr` yields `7000` which is the same as referencing `integerPtr` directly.
    - `&*integerPtr` ➔ `*integerPtr` yields `7` and `&*integerPtr` yields `7000` which is the same as referencing `integerPtr` directly.
    - `*&integer` ➔ `&integer` yields `7000` and `*&integer` yields `7` which is the same as referencing `integer` directly.
  - The net result is that these operations cancel each other out regardless of their order.

Illegal Assignments

- Careful with illegal assignments:
  - E.g.:
    - `int a = 7, *aPtr = &a;` — **Correct.**
    - `float b = 10.0, *bPtr = &b;` — **Correct.**
    - `&a = bPtr;` — **Attempts to change the address of `a`.**
    - `bPtr = a;` — **Attempts to store a non-address value into a pointer.**
    - `*aPtr = bPtr;` — **Attempts to store an address into a variable of type `int`.**
    - `*aPtr = &a;` — **Attempts to store an address into a variable of type `int`.**
    - `aPtr = *bPtr;` — **Attempts to store a non-address value into a pointer.**

Example – Swapping Two Integers

- Write a program that uses pointers to swap two integers.