LBYEC72 Lecture 2

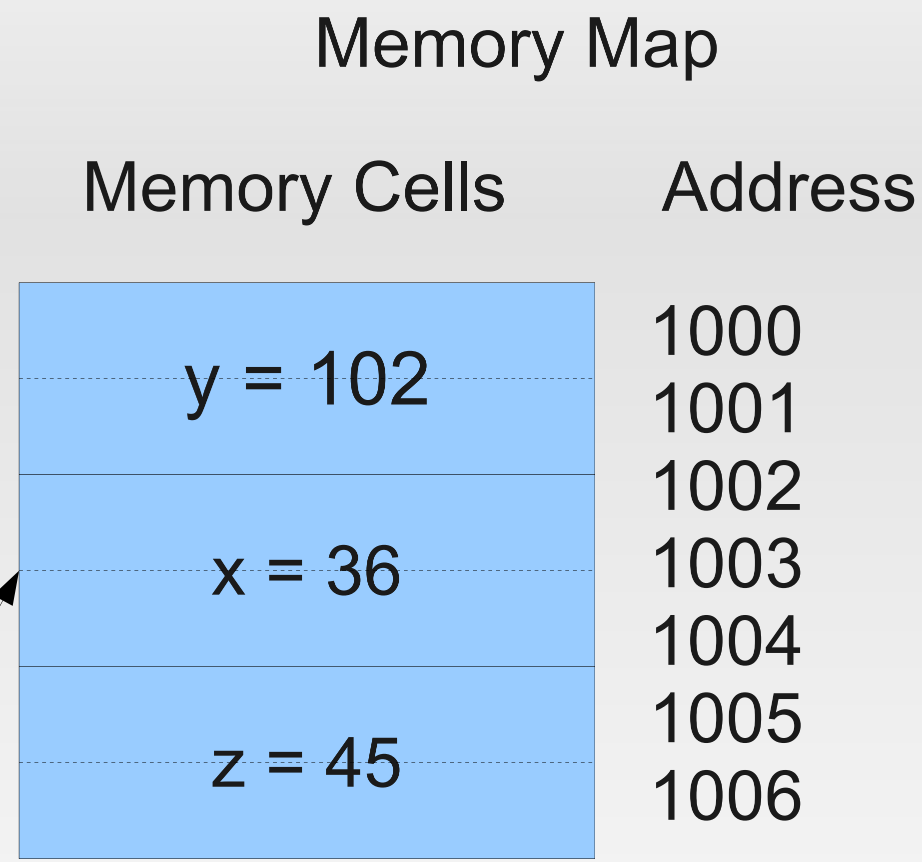
Pointers
Functions and Pass-By-Value
Functions and Pass-By-Refernce
Structures and Structure Arrays
Structure Pointers

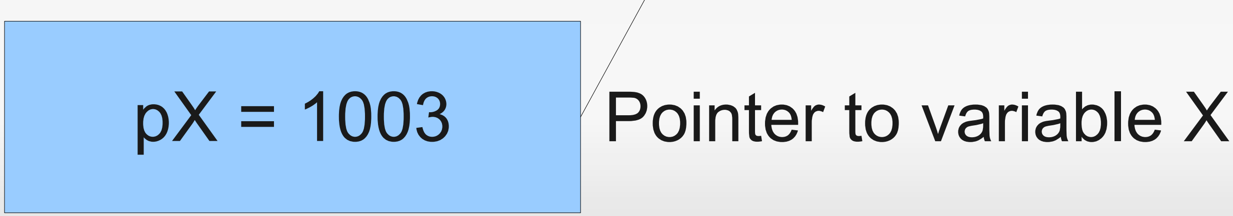
Pointers

- Pointers are a very powerful tool in C programming.
- It allows the programmer to access the contents of a variable using its address.
- Pointers are used for very fast passing of parameters to functions.

Pointers

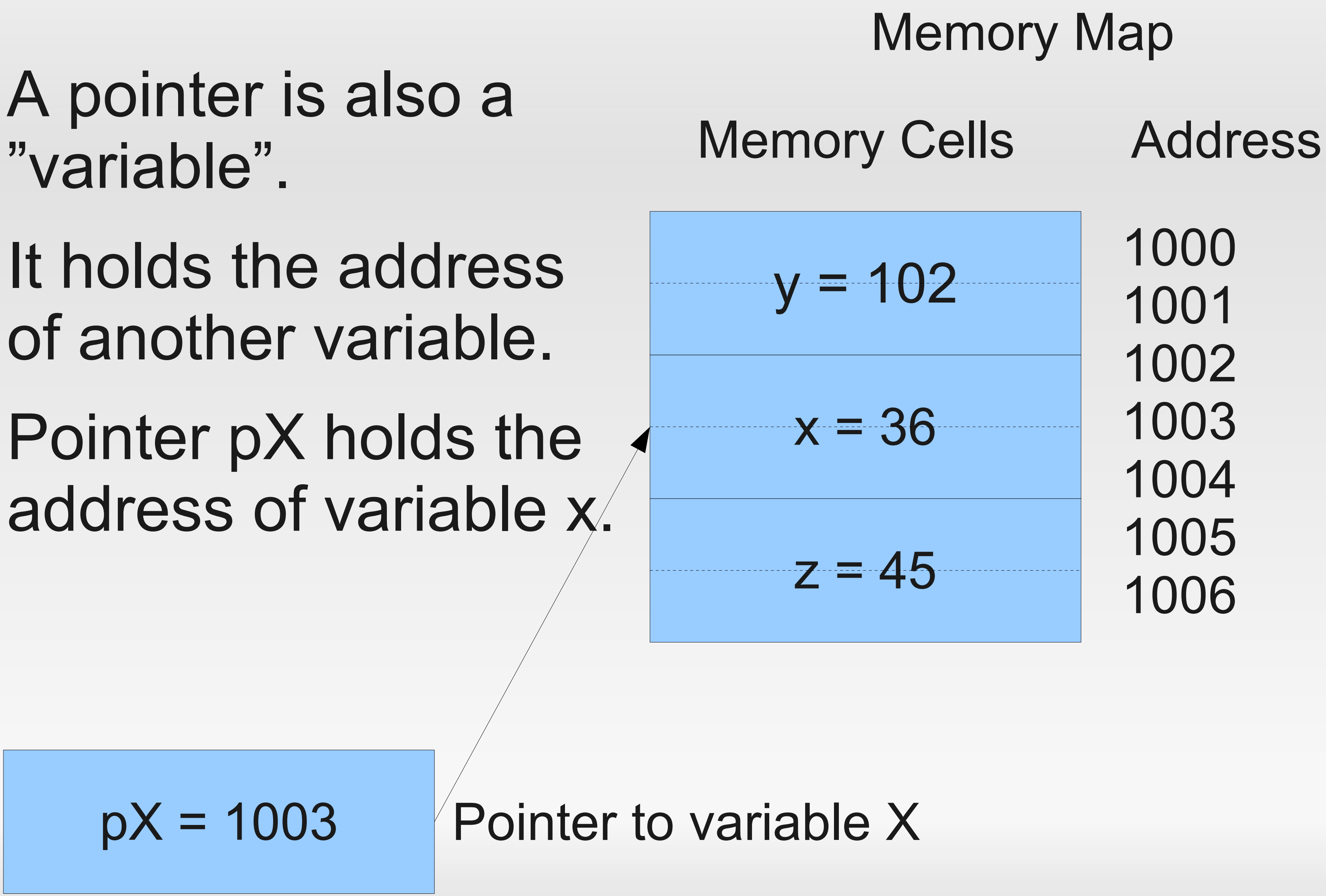
- Memory is divided into memory cells with 1 byte each.
- Each cell has an address.
- An integer is two bytes, so it occupies two memory cells.





Pointers

- A pointer is also a "variable".
- It holds the address of another variable.
- Pointer pX holds the address of variable x.



Pointers

- Think of a pointer as a cellphone number or a house address.
- The pointer can be used to access the variable ”indirectly.”

Pointers

- The create and declare a pointer:

```
int* pX;
```

Data type. Can be int, char, double, etc.

Indicates pointer

Name of pointer

Pointers

- To make a pointer point to a variable:

```
pX = &x;
```

Pointer

Variable

AddressOf operator. (&)
Gets the address of the
variable

Pointers

- Make sure that pointers always point to a variable.
- Pointers that are not pointing to variables would have garbage values and can cause errors.

Pointers: Exercise

- Consider the memory map example shown earlier. Create pointers that will point to variable y and z.
- What are the values of the pointers?

Dereferencing

- Values of variables can be indirectly accessed using pointers using the concept of dereferencing.
- To do derefencing:

`*pX`

Pointer

Dereference operator (*)

Dereferencing

- Consider another integer variable, myNum.
- If we use dereferencing on pointer pX to get the value of x ”indirectly”, we can assign the value of x to myNum.

```
myNum = *pX;
```

is the same as:

```
myNum = x;
```

Dereferencing

- We can also use dereferencing to indirectly set the value of x.

```
*pX = 3;
```

is the same as:

```
x = 3;
```

Dereferencing

CAUTION

Use dereferencing only if the pointer is pointing to an existing variable. If the pointer is not pointing to a variable, an error will occur.

Dereferencing: Exercises

- Consider the memory map example shown in the previous slides. Use pointer dereferencing on pointers to y and z to change the value of variables y = 10, z = x.

End of Lecture on Pointers

Functions

- Functions are blocks of code that can be called anywhere in the program.
- Consider the function printf() in stdio.h. The process of displaying text is arguably not just one line of code.
- Consider the sin() function in math.h. We can conclude the the computation of sine is made up of multiple lines of code.

Functions

- Functions are used extensively in modular programming. There are two reasons for creating functions:
  - Divide and conquer approach. Using functions allow the programmer to subdivide a big program into smaller functional units for ease of debugging.
  - Modularity and Reuse. Codes that are repeatedly used can be placed in functions for reuse later on.

Functions

- The use of functions is extensively used in modular programming style.
- Modular programming is a programming style that places pieces of code that are repeatedly called into functions.
- Consider the printf() function. Displaying text is a commonly used feature, so the printf() function was created to serve this purpose.

Functions

- A function is written in C programming this way:

Function Name

Parameter List Enclosed by Parenthesis

```
int addNumbers(int a, int b) {
    int temp;
    temp = a + b;
    return temp;
}
```

Return Value

Return Type

Start and End of Function Body Using Braces

Functions

- Return Type: Defines the data type that the function will return. Can be int, float, double, etc. For functions that do not return values, use void.
- Function Name: Any name can be used for the function as long as it follows the naming convention of C (similar to variable naming).

Functions

- Parameter List: Defines the parameters needed by the function. This is similar to the use of parameters in an algebraic function: for a function $f(x,y) = x + y + 3$, x and y are variables needed compute for the value of a function.
- Return Value: The actual value returned by the function. Should have the same data type as its return type.

Functions

- Start and End of Function Body: Defined using an Open Brace {, and a Closing Brace }, respectively. All functions codes should be enclosed by the braces.

Functions

- Some other notes about functions:
  - You can declare additional variables in the function body.
  - Variables declared in a function are local to the function, that is, no other functions can see those variables. These are called local variables.
  - Variables in the parameter list can be used inside the function body and are treated as local variables.

Functions

- Functions are run by ”calling” them in the main program (or other functions) using this notation:

```
int main() {
  int num;
  num = addNumbers(3,4);
  printf(”%d”, num);

  return 0;
}
```

Function Call

Note: You already called functions before by using printf() and scanf().

Functions

- Some other notes when calling functions:
  - You can also pass variables to functions aside from constant values like the previous example.
  - You can also ”not assign” the return value of a function to a variable by just simply calling the function, for example: intNumbers(3,4);. This is often done with functions declared as void, like printf() and scanf().

Functions

- When declaring functions:
  - Write functions before the main program. The main program will not be able to see the function if it is written below the main program.
  - An exception to the rule above is when a **function prototype** is declared above the main program.

Functions: Exercises

- Create a function that will display Hello World! on screen. Call the function in the main program.
- Create a function that will compute for the area of a circle. Use the function in the main program by setting fixed (hard-coded) values of radius. Show the radius and the computed area.

Functions: Pass-By-Value

- There are two ways of passing values to functions:
  - Pass-by-value
  - Pass-by-reference (also called pass-by-address in some literatures)

Functions: Pass-By-Value

- Pass-by-value copies the values of variables or constants passed through the parameter list.
- Consider the previous example function, addNumbers():

```
int addNumbers(int a, int b) {
    int temp;
    temp = a + b;
    return temp;
}
```

Functions: Pass-By-Value

- The values of variables passed to the functions are copied to a and b.

```
x = 3;
y = 4;
z = addNumbers(x, y);
```

The value of x is copied to a.
The value of y is copied to b.

Functions: Pass-By-Value

- When addNumbers() compute for the sum of a and b, it no longer needs x and y. Only the local variables, a and b, can be seen by addNumbers().
- This safeguards x and y from being modified by addNumbers().

Functions: Pass-By-Reference

- Sometimes we need to **intentionally** modify the values of variables outside the function.
- Consider the swapNumbers() function below:

```
void swapNumbers(int a, int b) {
    int temp;
    temp = b;
    b = a;
    a = temp;
}
```

Functions: Pass-By-Reference

- The function swapNumbers() swaps the value of a and b. If a = 3 and b = 4, a = 4 and b = 3, after swapping.
- Consider this code running in the main program:

```
x = 3;
y = 4;
swapNumbers(x,y);
```

Functions: Pass-By-Reference

- Based from the code of swapNumbers(), we can say that x and y should be swapped.
- However, since x and y were **copied** to a and b, it was actually a and b that was swapped, not x and y.
- In the end, x and y would still have the same values.

Functions: Pass-By-Reference

- To manipulate variables outside the function, we need pass-by-reference.
- Pass-by-reference works by passing the reference, or address, of the variable to the function, instead of passing its value.
- Having the address of the variable allows us to manipulate it through the process of dereferencing. (see Pointers slides)

Functions: Pass-By-Reference

- To take advantage of pass-by-reference, the new swapNumbers() function will now look like this:

```
void swapNumbers(int* pA, int* pB) {
    int temp;
    temp = *pB;
    *pB = *pA;
    *pA = temp;
}
```

Take note of the use of the dereference operator (*).

Functions: Pass-By-Reference

- The following changes were done in swapNumbers():
  - Instead of having variables as parameters, swapNumbers() now have pointers as parameters.
  - Instead of manipulating variables, swapNumbers() indirectly manipulate the variables pointed by the pointers through the use of dereferencing.
  - When swapNumbers() end, the variables that it manipulated become swapped.

Functions: Pass-By-Reference

- The codes in the main program should also change:

```
x = 3;
y = 4;
swapNumbers(&x, &y);
```

Functions: Pass-By-Reference

- The following changes were done in the main program:
  - Instead of passing the values of x and y to swapNumbers(), we passed the **address** of x and y instead.
  - The address can be retrieved using the AddressOf(&) operator (see Pointers slides).
  - At the end of swapNumbers(), the values of x and y become swapped.

Functions: Pass-By-Reference

- Pass-by-reference can be intimidating at first. However, its use offers a lot of advantages.
- Pass-by-reference is fast. Pass-by-value is slow in comparison, as it has to copy the values to variables. (Which is faster, copying a double value, which is 8 bytes, or an address value, which is only 2 or 4 bytes?)

Functions: Pass-By-Reference

- Pass-by-reference allows functions to manipulate variables outside of its body. This is very useful for creating modular and reusable functions.
- Pass-by-reference are used extensively when manipulating data structures.

Functions: Pass-By-Reference

- Actually, you've used pass-by-reference before.
- Consider an example code using scanf():

```
scanf("%d", &x);
```

Functions: Pass-By-Reference

- Since scanf() needs to manipulate a variable outside its body, we need to pass the address of the variable as its parameter. This is actually pass-by-reference.
- This is the reason for appending the umpersand (&) before the variable.

Functions

End of Functions Presentation

Structures

- Recall: arrays are very useful for storing similar pieces of data.
- Examples:
  - Storing and manipulating 10 integers.
  - Storing and manipulating 3 double values for use in computations.
  - Storing and manipulating characters and treat them as a string.

Structures

- Arrays have a limitation, however. They cannot efficiently store data with different data types.
- Consider this example: Store data about students. Each student would have a name and an ID number.

Structures

- Using arrays, we can store data about students this way:

```
char names[10][128];
int ids[10];
```

Structures

- Using multiple arrays to store data about students can be tedious.
- If the number of students changed, then all array sizes should be updated, and so are the codes that work with the arrays.
- It is more convenient for the programmer if we could group the data about the student in one place.

Structures

- A structure organizes data that are coherent with each other.
- A student with name and id number is easier stored in a structure.

Structures

- To declare a student structure:

Structure Name

```
struct STUDENT {
  char name[128];
  int id;
};
```

Keyword struct,
Used to declare a structure

Attributes of Structure

Structures

- A structure is created like a variable in the main program:

Name of Structure

```
struct STUDENT myStudent;
struct STUDENT student2;
```

Keyword struct,
Used to create
structure variable

Variable Name
For Structure

Structures

- Accessing attributes of structures require the use of the dot ”.” operator.

```
student2.id = 100;

strcpy(myStudent.name, ”John”);

printf(”id of student:%d”,
        student2.id);
```

The structure variable precedes the dot.
The attribute then follows the dot.

Structures

- Structure attributes are treated just like any variables.

CAUTION

Only attributes declared in the structure declaration can be accessed using the dot operator.

Structures: Exercise

- Create a structure for an Employee. The employee has the following attributes: name, id number, age.
- Create a program that will set the attributes of the Employee and display the set attributes.

Structure Arrays

- Like creating integer and double arrays, we can create a structure array by using the bracket notation:

```
struct STUDENT students[10];
```

This creates 10 students in memory.

Structure Arrays

- Accessing each element in the structure array also requires the dot notation.
- The difference for arrays is that we first make use of the brackets to select what element of the array we want to access.

Access student 0. (1st element)

```
students[0].id = 100;
printf("%s", students[1].name);
```

Access student 1. (2nd element)

Structure Arrays

- Like arrays, sructure arrays are very powerful since we can use iteration on each element to allow manipulation of hundreds and thousands of elements.

```
for(ctr=0;ctr<10;ctr++) {
  students[ctr].id = 0; // Clear ID
}
```

Structure Arrays: Exercise

- Modify the exercise from Structures slides making use of Employees to set the id number of 10 students from 0 to 10. The name should be ”none” (no quotes).
- Show each employee element in the array.

End of Structures and Structure Arrays

Structure Pointers

- Like pointers, structure pointers provide the advantage of quickly passing structures to functions.
- Structure pointers also allow functions to manipulate structure outside of its scope (non-local variables)
- This is a very important concept for creating Abstract Data Types (ADTs not discussed).

Structure Pointers

- To create a structure pointer:

Structure Name

Structure Pointer

```
struct STUDENT* pStudent;
```

Keyword struct, used to create structure pointer.

Operator for specifying creation of structure pointer

Structure Pointers

- Similar to pointers, structure pointers should point to valid variables in memory.

```
pS = &myStudent;
```

Valid Structure Variable
(declared earlier)

Structure Pointer

AddressOf (&)
Operator

Structure Pointers

- Similar to pointers, the structure variable being pointed by the structure pointer can be indirectly accessed through the pointer using dereferencing.

`*pS` Is just like -> `myStudent`

Dereference

If pS = &student2;

`*pS` Is just like -> `student2`

Structure Pointers

- Dot notation is also used to access the attributes through a structure pointer.
- Dereferencing should be done first before using the dot notation.

```
(*pS).id = 0;
printf("%s", (*pS).name);
```

Structure Pointers

CAUTION

Use dereferencing only if the pointer is pointing to an existing variable. If the pointer is not pointing to a variable, an error will occur.

Structure Pointers: Exercise

- Modify the exercise from Structures slide. Create an additional Employee structure pointer and set the values of the Employee variable through the structure pointer.
- Display the contents of the Employee variable through the structure pointer.

Structure Pointer

- The dereferencing and dot notation for a structure pointer can be tedious at times.
- A shortcut and more elegant way of doing it is using the arrow notation:

```
ps->id = 0;
printf("%s", pS->name);
```

Using typedef

- The keyword typedef can be used to simplify the declaration of structures and structure pointers.
- typedef is actually used to create custom data types.
- The use of typedef will be limited to structures in this lecture.

Using typedef

- To use typedef in structure declaration:

Keyword typedef

Keyword struct

```
typedef struct {
  int id;
  char name[128];
} Student_t;
```

Body of structure

New custom data type

Using typedef

- To declare a structure variable and structure pointer using the new custom data type:

```
Student_t myStudent;
Student_t* pS;
```

Notice that the new custom data type effectively becomes just like int, double, and other data types.

End of Structure Pointers Lecture