# ICOM 4035 – Data Structures

Dr. Manuel Rodríguez Martínez

Electrical and Computer Engineering Department

Lecture 2 – August 23, 2001

# Readings

- Read Appendix D of textbook
  - Primitive Arrays in C++
- Read chapter 1 of textbook
  - Arrays, pointers and structures
    - Do not read section 1.6.3

# Built-in Arrays in C++

- Arrays are one of the most fundamental constructs in any programming language.
- An array is a collection on N elements of the same data type.
- C++ array declaration:

  int size=5;

  int nums[size];

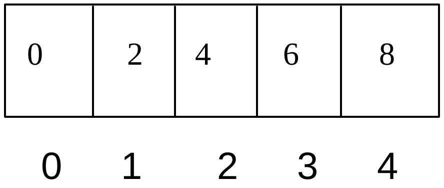
  - Array of 5 elements
  - Run-time support system will provide block of contiguous memory large enough to accommodate it.
- New C++ standard comes with new array type
  - Vector – is a class with more features than basic array

# Built-in arrays in C++ (cont.)

- Suppose an array is initialized as follows:

```
for (int i=0; i < size; ++i){
    nums[i]= i * 2;
}
```

- Result of this will be:



| | | | | |
|---|---|---|---|---|
| 0 | 2 | 4 | 6 | 8 |
| 0 | 1 | 2 | 3 | 4 |

- Can have default initializers
  - char vowels[] = {‘a’, ‘e’, ‘i’, ‘o’, ‘u’};
  - Initializes the array with 5 elements, each with the corresponding letter.

# Built-in arrays in C++ (cont.)

- Compiler computes the space for the array when the default initializer is used.
- Arrays can be arguments to functions.
- A string is basically an array of char:
  - char name[] = “Manuel”;
  - This is equivalent to:
    - char name[] =  {‘M’, ‘a’, ‘n’, ‘u’, ‘e’, ‘l’, ‘\0’};
    - The ‘\0’ is the end of string character, thus the string has 7 characters, including the ‘\0’.
  - New C++ standard has a new string class with lots of features.

# Multi-dimensional arrays

- Multi-dimensional arrays:
  - int table[2][3] – two-dimensional array of with 2 row and 3 columns, for a total of 6 entries.
  - Addressing is more complicated
    - Will need to variables to index elements
      - table[i,j]
    - table[0,2] – access element on row 0, column 2

| | | |
|---|---|---|
| [0,0] | [0,1] | [0,2] |
| [0,1] | [0,1] | [0,1] |

# Function Calls: Call By Value

- In call by value, the value of the parameters are copied into temporary variables which are accessed by the statements in the body of the function.

```
int sqr(int x){
    return x * x;
};
. . .
int y = 2, m = sqr(y);
```

  - The values of the parameter cannot be modified in the body of the function

```
int sqr2(int x){
    x *= x;
    return x;
}
…
int y = 2, m = sqr2(y);
```

    - **The value of y still is 2 after the function call.**

# Function Calls: Call ByReference

- In call by reference, a reference to the memory addresses of the parameters is passed to the statements in the function body. Avoids extra copy of values!

```
int sqr(int& x){
    return x * x;
}
…
int y = 2, m = sqr(y);
```

  - The values of the parameter can be changed within the body of the function.

```
int sqr2(int& x){
    x*=x;
    return x;
}
…
int y = 2, m = sqr2(y);
```

    - **Now, the value of variable y has become 4**.

# Function Calls: Call By Constant Reference

- In call by constant reference, a constant reference to the memory addresses of the parameters is passed to the statements in the function body. Avoids extra copy of values!

  ```
  int sqr(const int& x){
      return x * x;
  }
  …
  int y = 2, m = sqr(y);
  ```

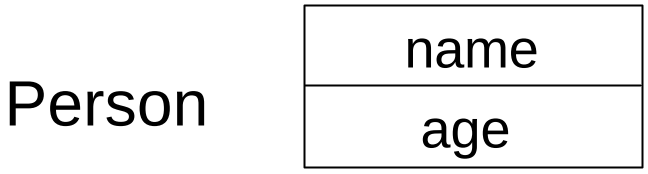
  - The values of the parameter cannot be changed within the body of the function.

    ```
    int sqr2(const int& x){
        x*=x;
        return x;
    }
    …
    int y = 2, m = sqr2(y);
    ```

    - **The value of variable y remains 2 after the function call**.

# Structures in C++

- Fundamental to build complex data structures.
- Provide the mechanism to represent multiple data values with a single data type.
- Consider a Person data type consisting of a name and age fields. Conceptually, we have:



Person

name

age

- In C++ we can write:

```
struct person{
        char[100] name;
        int age;
}
```

# Using Structures in C++

- We can declare variables based on the types for the structures:

```
struct person{
   char[100] name;
   int age;
}


struct person student;
```

  - Now the variable student can be used to hold information about some person, which appear to be a student.
- Individual fields of the structure are access using the dot notation:
  - student.name – gives access to the name field
  - student.age – gives access to the age field.
  - Convention: <variable name>.<field name>

# Typedef and Structures

- It is a good idea to create a type name for each structure because it makes your code more readable.
  - Use the typedef command for this purpose

```
typedef struct person{
   char[100] name;
   int age;
} person;

person student;
```

  - No need to add the word struct anywhere in the code

# Structures and functions

- Structures can be passed as arguments to functions
  - – Use call-by-reference!
    - pass the address of the structure
  - – Call-by-value involves copying the whole structure into a temporary variable, thus there is too much overhead.
  - – Examples

```
void printPerson(person & thePerson){
   // call-by-reference  ☺ - efficient
   …
}

void printPerson(person thePerson){
   // call-by-value ☹ - inefficient, too much copying
}
```

# Structures and Functions

- Return pointers to structures as the return value from a function.
  - Pointer created by calls to new operator.
- Returning the structure by value would be inefficient because the whole structure must be copied to a temporary variable.
- Do not return references or pointers to structures local to the function because these cease to exits upon return from the function.
  - Big bad bug in your program!

# Structures and Functions (cont.)

- Examples:

```
person *makeNew(char[] name, int age){
    person *result;
    result = new person;
    result->name = name;
    result->age = age;
    return result;
}
```

- This returns a pointer to the structure.
- Usually the way to go!
- The notation -> replace the dot (.) notation since now the variable is a pointer.

# Structures and Functions (cont.)

- This mechanism is correct but somewhat inefficient

```
person makeNew(char[] name, int age){
    person result;
    result = new person;
    result.name = name;
    result.age = age;
    return result;
}
```

- The return call will cause the value of result to be copied to another variable.
- If the structure has many fields, this would be very bad.

# Structures and Functions (cont.)

- This mechanism is incorrect

```
person *makeNew(char[] name, int age){
    person result;
    result = new person;
    result.name = name;
    result.age = age;
    return &result;
}
```

- Variable result is destroyed upon the return call, so its address will go away. Thus, the pointer returned will be a pointer to nowhere.
  - This is a big bad bug!

# Rules of thumb for Structures

- Pass structures by reference, constant reference or as pointers to the functions.
- Return structures by values if they are small.
- Return pointers to structures as result of function calls if the structures are big.

# Pointers in C++

- What is a pointer?
  - A pointer is a variables that stores the address of a memory location of in which the data for another variable or object is stored.
  - A pointer can “point” to data that belongs to as simple variable (e.g. an int), an array, a structure or an object.
- Why do we need pointers?
  - Sometimes we cannot predict how many variables, or how much memory we might need to run our program.
  - Pointers provide the mechanism to allocate and deallocate memory in a dynamic fashion.
- Are we going to use pointers a lot in this course?
  - Yes.

# How big are pointers?

- All pointer are of the same size, since their value is an integer number that represents a memory location (a memory address).
- The size of a pointer will depend on the architecture of the underlying computer.
- A 32-bit architecture like Intel Pentium II has pointers of 32-bits (4-bytes).
- A 64-bit architecture like Sun Ultra SPARC has pointers of 64-bits (8-bytes).
- Typically the name of the locations addressed by pointers are termed “words”.

# Words? What are Words?

- Words are the minimal units of memory that the CPU can retrieve from the pool of bytes available in main memory.
- Why is this thing done?
  - Because it is inefficient for the CPU to be interrupted to bring just 1-byte. Therefore, computer architects and engineers came up with designs in which bytes were fetched from memory in groups. These groups are called words.
  - A 32-bit architecture brings bytes in groups of 4.
  - A 64-bit architecture brings bytes in groups of 8.
- Then, why can we have things like char, and short which are smaller than words?
  - The run-time system takes care of hiding the memory alignment from the programmer.
  - Structures, however, bring this ugly issue to the surface…

# Declaring pointer variables

- To declare a pointer variable, you must write the * symbol before the name of the variable:

```
int x = 10, y = 20;  // regular variables
int *p; // pointer to an integer
```

  - Initially a pointer has no defined value, thus it points to an undefined memory location (BUG In Progress!)
  - One can initialize a pointer to a given location:

```
int *p= &x; // gives p the address of x,
```

    - Now p points to the location in which x is stored.
  - Changes made to a location via a pointer behave like changes made via the variable associate with the location.

```
*p = 40;  // now variable x has value 40
```

  - The notation *p is used to access the value pointed by p.
  - In this case, the * is called the dereferencing operator.

# NULL pointer value

- Since a pointer initially points to an undefined location, it is always a good idea to initialize it to a well-known yet illegal memory location. This memory location is the 0 address, and in C++ we have a name for it: NULL.
- Golden Rule of Pointers:
  - If you don’t know what address to give a pointer at the moment of declaring it, then initialize it to NULL.
  - Example:

```
int x = 10, y = 20;
int *p = NULL; // safest thing on the planet
```

# Memory allocation with new

- To allocate a well-known and valid memory location to a pointer, you must use the new operator.

```
int x = 10, y = 20;
int *p = NULL;
p = new int;
*p = x + y; // gives *p the value 30
```

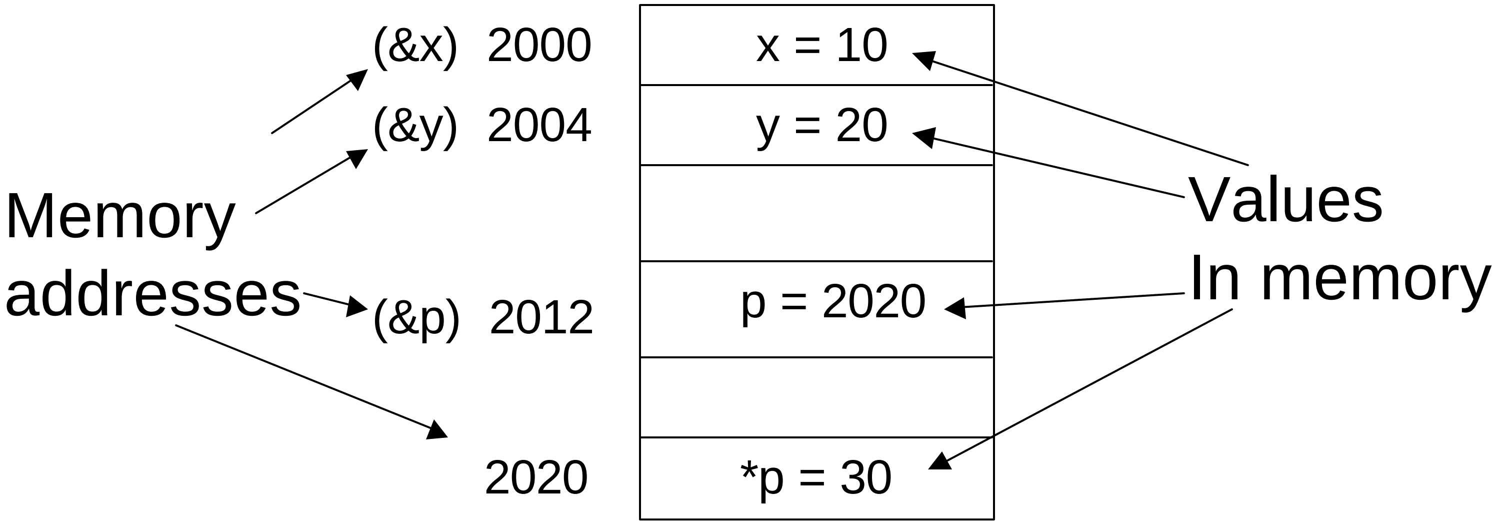
  - It is also possible to initialize the value of the location pointed to by the pointer via the new operator:

    ```
    p = new int (17);
    ```

    - This statement make *p equal to 17.

# Memory model: p vs. *p

- Remember that the value of the pointer is a memory location. The value of the "thing" pointed to by the pointer is the value stored at that memory location.
- In our example, p is a memory location, and *p is the value stored at the location pointed to by p.


(&x) 2000
x = 10
(&y) 2004
y = 20
Memory
addresses
(&p) 2012
p = 2020
2020
*p = 30
Values
In memory