ICSI 403
Design and Analysis of Algorithms

Lecture 03 – Introduction to C++ Programming, part 2

J Marques de Carvalho

Last Time

- C++ Basics
  - History of C++
  - Identifier naming rules
  - Data types
  - Variable declarations, initializations, constants
  - Arithmetic operators and expressions
  - Using `cin` and `cout`
  - Comments (`//` and `/*..*/`)

Last Time

- Flow of Control
  - Boolean operators and expressions
  - Logical operators and precedence
  - Short-circuiting
  - Non-zero integers are `true`; only zero is `false`
  - `if-then` and `if-then-else`
  - Code blocks in braces
  - `switch` statement
  - `enum` types
  - The conditional operator `cond?value1:value2`
  - Loops – `for`, `while`, and `do…while`
    - `break` and `continue`

Last Time

- Functions
  - Library functions
  - `#include` directive
  - `<>` vs `""` on `#include` files
  - Preprocessor directives (like `#include`)
  - `void` functions
  - `exit()`
  - `rand / srand` for random numbers
  - Declaring functions with a function prototype
  - Parameter lists (can be () – empty)
  - `main` is an `int` function. Return 0 if program ran OK
  - Recursion (just like Java)
  - Scoping rules

Parameters and Overloading

- Java passes everything *by value.*
- Consider primitive types (`int`, `float`, ...)
- The method called receives a `copy` of the value passed.
- If the method called modifies a parameter, the new value is lost when it returns to the caller.
- Java passes only *objects* by reference (object variables are references to the object).

Parameters and Overloading

- C++ gives us more flexibility.
- We can pass simple variables by value (default).
- We can pass them by reference.
- To indicate a pass-by-reference parameter, append an ampersand (&) to the variable's type

```
void swap(int& a, int& b);
```

- Pass-by-value and pass-by-reference can be done on a parameter-by-parameter basis.
- Using pass-by-reference allows, in effect, a function to return more than one value.

Parameters and Overloading

- In `main()`…

```
int a=1, b=2;
cout << a << b;
swap(a, b);
cout << a << b;
```

```
void swap(int& a, int& b)
{                     // swap doesn't work with
    int c = a;        // copies of a and b;
    a = b;            // It gets references to
    b = c;            // (the addresses of)
}                     // of its operands.
```

Parameters and Overloading

- Because the call*ed* function receives the address of the variable in the parameter, the call*ing* function can't use an expression or a constant in the parameter list; it must use a variable.
- The ampersand can follow the type or precede the parameter.
- You'll see it done both ways:

```
void swap(int& a, int& b)
void swap(int &a, int &b)
```

Parameters and Overloading

- C++ supports the overloading of functions.
- The function must be of the same return type and have differing numbers and/or types of parameters.
  - I.e., C++ differentiates the two by parameter list.
  - Call-by-reference vs call-by-value is not enough to differentiate.

Parameters and Overloading

- C++ resolves overloading (which one to use?) by
  - If there's an exact match (number and type of arguments, use THAT).
  - If not, then if there's an exact match using automatic type conversion (example: calling function passed an `int`, called function expected `double`), then use that.
  - This covers almost all situations you'll encounter.

Parameters and Overloading

- C++ supports default parameters on functions.
- Applies only to call-by-value.
- Can be done using overloading, but when it applies, it's shorter to use default parameters.
- We indicate the default parameters on the function prototype in the function declaration.

Parameters and Overloading

```
void showVolume(int length, int
  width=1, int height=1);
```

- We can call `showVolume` with three `int`s, (like any other function call), or with two `int`s (in which case it will use 1 for `height`), or one `int` (in which case it will use 1 for the `width` and 1 for the `height`).
- If you're going to omit parameters, they must be omitted from the right end.
- In this example, there's no way to omit `width` without also omitting `height`.

Parameters and Overloading

- Preprocessor directives:
  - We've already seen #include
  - Another frequent use is for conditional compilation
  - `#define` defines values that are only visible at compile time.
  - These values can then be used to select which code to include in your project.
  - `#define TEST` sets "TEST" to true
  - `#if / #else / #endif` can then use TEST to tell what code to select for inclusion at compile-time
    - This is different from using "regular" `if-else` – the clause not taken (then or else) isn't even compiled.

Parameters and Overloading

```
#define TEST 1          (or #define TEST 0)
…
#if TEST
<this code will be compiled>
#else
<this code will not be compiled>
#endif
```

- This can be very convenient for turning on and off debugging statements.
- It doesn't determine whether they *execute* or not; it determines whether they'll even *exist* or not.

Parameters and Overloading

- Assertions are related to preprocessor directives.
- As in Java, an assertion is a condition that should always be true in your program; if it's false, something is wrong
- Assertions can be used during debugging to see what conditions hold at various points in your program

Parameters and Overloading

- To use assertions, we `#include <cassert>`
- Then you can code an assertion:

```
assert (condition);
```

- If `condition` is false, your program will stop with an error
- To turn off all assertions (i.e., strip them from your compiled code, while leaving them in your source file), put `#define NDEBUG` somewhere before `#include <cassert>`
- Personally, I prefer if/print statements for debugging.

Arrays

- Arrays in C++ are handled differently than in Java
- They're not objects; they're multiples of primitive types.
- We use square brackets to denote subscripts.
- They're zero-based, like in Java.
  - A[0] is the first element.
- Arrays must be declared.

Arrays

```
int x[10];
int x[7] = {12, 7, 4, 9, 6, 8, -1};
int x[]  = {12, 7, 4, 9, 6, 8, -1};
```

- You can't use a variable for sizing an array; you must know the size at compile time (for one declared this way; we'll see dynamic arrays later)
- You *can* use a constant, though:

```
int x = 10;
double y[x];            // illegal

const int x = 10;
double y[x];            // legal
```

Arrays

- When we pass an (entire) array as a parameter, we pass the address to the start of the array.
- Because the array isn't an object, it's implemented differently than in Java, but the behavior is the same:
  - The called function gets access to the whole array (read and write).
- By default, *passing a single array element is done by value*, although it can be passed by reference, in which case the called function can modify that one value.

Arrays

```
int arraySum(int values[], int count);

int main()
{
   int A[3] = {0, 1, 2};    // create array A
   cout << arraySum(A, 3); // output the sum
   return 0;
}
int arraySum(int values[], int count)
{
    int sum = 0;                   // initialize sum
    for(int i=0; i<count; i++)  // iterate
        sum+= values[i];           // add to sum
    return( sum );
}
```

Arrays

- Because arrays are not objects, they don't have properties, including `.length`, or number of dimensions – they're just the address of a block of memory.

  `for (i=1; i<arrayname.length; i++)` doesn't work!

- Generally, you should pass the array's size as another parameter – the called function has no way of knowing how big the array is.

- When we pass an array, we just pass the array name, not the brackets.

Arrays

- `const` arrays.
  - When we pass an entire array, the called function has read / write access to the whole array.
  - If we're counting on the called function to not change anything in the array (i.e., it only needs read access), then we're left unprotected.
  - `const` to the rescue!
  - Declaring the array parameter as `const` will enforce the passing of the array as read-only – the called function will not be allowed to modify any array elements.
  - `void processArray(const int a[], int sizeOfA)`

Arrays

- Because arrays aren't an object, we can't write a function that returns an array per se.
- There is a way around this, but it involves pointers, which we haven't covered yet.
- Just know that we can't have a function declared of type `int[]`:

```
int[] getValues(int count)
{
        …
}
```

is illegal

Arrays

- Multidimensional arrays have multiple sets of brackets: `int a[3][3];` creates a 3-x-3 array
- When you pass a multidimensional array as a parameter, the called function must declare the sizes of all dimensions but the first:
- In `main()`:

```
    char page[30][100];
    showPage(page, 30);
    …

void showPage(char page[][100], int size)
{
    for (int i=0; i<size; …
```

Multidimensioanl Arrays

```
#include <stdio.h>
const int N = 3;

void print(int arr[][N], int m)
{
    int i, j;
    for (i = 0; i < m; i++)
      for (j = 0; j < N; j++)
        printf("%d ", arr[i][j]);
}

int main()
{
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    print(arr, 3);
    return 0;
}
```

Pointers (briefly)

- Let's jump out of order just a bit and look at pointers (briefly).
- Pointers, the fact that they exist, and how they are handled are one of the biggest differences between C/C++ and Java.
- A <u>*pointer*</u> is the address of something.
- Java has something similar in reference variables, but they're rather limited in use.
  - A (Java) reference variable can only refer to an object
  - A (C++) pointer can refer to anything, or to nothing in particular

Pointers (briefly)

- We declare pointers just like we declare any other variable (after all, the compiler must set aside space to store the address the pointer points to).
- But we must tell what kind of data the pointer will point at.
- We indicate a pointer with an asterisk:

```
int *a;
```

- In this example, `a` is a pointer that points to (holds the address of) an integer.

Pointers (briefly)

- We get the address of something with the ampersand (&).
  - We've already done this with passing variables by reference.
- For some variable `X`, `&X` is "the address of `X`“.
- For some pointer `p`, `*p` is what `p` points to (the value stored at the address that's in `p`).
- Using `*p` (where `p` is a pointer) is called *dereferencing the pointer*, and the value of a dereferenced pointer is just the value the pointer points to.

Pointers (briefly)

- Some sample pointer operations:

```
int x, *p;    // x is an int; p is a pointer to an int
x = 1;        // give x a value
p = &x;       // p holds x's address (p points to x)
*p = 3;       // 3 is stored at the address in p (x==3)
(*p)++;       // the int p points to is incremented
cout << x;    // the output will be…?
```

Pointers (briefly)

- Just as with Java's reference variables, two pointers that are equal point at (refer to / have the address of) the same thing:

```
int x, y;     // x and y are ints
int *p, *q;   // p and q can point to some int
p = &x;       // p points at (holds the address of) x;
q = &y;       // q points at (holds the address of) y;
x = 1;        // give them some values
y = 2;        //
p = q;        // now p and q both point to y
```

- If we want to see if two pointers contain the same address, we can compare them (`p == q`).
- If we want to see if what they point at hold the same value, we can use (`*p == *q`).

Pointers (briefly)

- We can do arithmetic on a pointer:

```
int array[100];
int *a;       // pointer to an int
a = &array; // a points at array ([0])
a++;          // a now points to A[1]
```

- Because `a` is a pointer to an `int`, incrementing `a` adds 4 to it; not 1 (`int`s are 4-byte values).
- When you increment a pointer, you don't "add 1"; you "point at the next one of these“.
- The system takes care of how much to actually increment by (the size of the value).

Pointers (briefly)

- Pointers are a full-blown data type.
- We can pass a pointer as a parameter to a function.
- A function can return a pointer to something.

```
int *getNextPointer(int *p)
```

Pointers (briefly)

- Dynamic memory allocation looks a lot more like Java, because we use the *new* keyword, and we get a pointer (reference) back from new:

```
int *a, *array;
a = new int;  // allocates an int and
              // constructs an int object
*a = 30;      // put something in it
delete a;     // destroy the integer
array = new int[100]; // create an array
array[3] = 7;         // put something in
delete[] array;       // destroy it
```

Pointers (briefly)

- Pointers that don't point anywhere are null.
- NULL (all caps) is a system-defined constant (not a C++ reserved word).

```
int *p=NULL;
p = &q;
```