ESS201: Programming-II
Module: C++

Jaya Sreevalsan Nair
jnair@iiitb.ac.in
International Institute of Information Technology, Bangalore

Term I: 2018-19; Lecture-03 on October 22, 2018

Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live

– attributed to John Woods.

1 Recap: Friends

Some functions have to be defined outside of a class interface. However, for some of them, according to the traditional definition of the class concept, those functions which *need not* be defined in the class interface itself should nevertheless be considered functions belonging to the class. That is, if permitted by the language syntax, they would certainly have been defined inside the class interface.

There are two ways to implement such functions.

1. Implementing those functions using available public member functions.
2. Applying the definition of the class concept to those functions, one can state those functions in fact belong to the class. Hence, they should be given direct access to the data members of objects. This is accomplished by the `friend` keyword.

2 C++ Classes

- **Memory management in constructors and destructors**: Usage of stacks and (memory) heap determine the environment/scope in which certain entities can exist. C++ uses the philosophy of *control of efficiency* for managing data for an object. For improving computation speed at run-time, the program is written in a way that the lifetime and storage of data is pre-determined.
  - Static memory allocation is managed using stacks. The stack is an area in memory that is used directly by the microprocessor to store data during program execution. Variables on the stack are called automatic or scoped variables. The static storage area is allocated before run-time, i.e. at *compile-time*. Using the stack or static storage area places a priority on the speed of storage allocation and release, which can be useful at times. The compiler decides *how long* the memory is allocated on the stack. However, static allocation reduces the flexibility in programming.
  - Dynamic memory allocation is managed through heap. The heap is a pool of memory which is determined at runtime. "How much" and "for how long" those "pieces" of memory are required are determined at run-time. The new objects are created on the heap, as required at run-time. Just as the programmer has the *flexibility* of creating objects on the fly, (s)he has the *responsibility*

to clean up as well after use. Thus, when the storage is not required any more, the memory on the heap must be released using *delete*. Alternatively, a *garbage collector* can be used to clear memory not in use.

As per design, dynamic allocation is relatively slower than static one, as the latter is equivalent to a single microprocessor instruction to move the stack pointer. Additionally, for dynamic memory management, since the tolerance for garbage collectors (GCs) is subjective to the program, GCs are not included in the language specification and hence, are provided by external programs/libraries.

3 Good Practices for Efficient Programming

Know the compiler and debugger well. Philosophies of debugging are as follows:

- Professional programmers make heavy usage of debugging tools.
- Importantly, debugging saves time and frustration.
- Fundamental principle of debugging is *confirmation.* The idea is to sequentially figure out until which point in your program the processes are going as per design, and which is the point at which it stops confirming to design. Thus, we *locate* where the bug is using a debugger, which is the precursor to the actuall debugging process.
- Do not use `printf`() or `std::cout` statements, as they need separate recompilation and re-execution cycles, and they also interfere with the program, at times. Overall, this practice is inefficient and less informative.
- Binary search principle: Check if everything is working as per design at the "approximate" halfway point in the program. If so, then check at the approximate $\frac{3}{4}$ -way point; if not, check at the $\frac{1}{4}$-way point. Each time, narrow down the location of the bug to one half of the previous portion. (*Hint: divide your program logically as opposed to its actual execution on its data.*)
- Learn to work with breakpoints and inspecting the variables.
- Fundamental Principle for Execution Errors: The first step to take after an execution error is to run the program through the debugger (if the error occurred when running it without the debugger), to determine where the execution error occurred. Where implies inside a loop, or in between loops, or within a branch, or within an iteration.
- Usual places of error:
  - Starting and ending conditions in for/while loops;
  - Segmentation errors when there is a buffer overflow – usually happens when accessing an index in an array/vector which has not been allocated; or when deleting null pointer or empty stacks.
  - Termination conditions do not *terminate* in actual implementation (i.e. when the program hangs) – usually happens in the while loop.

Learning debugging is a huge time-saver in professional software development. Learn first with `gdb` which is non-graphical debugger and progress to a debugger with graphical interface, e.g. DDD, KDB, etc. Several integrated development environments (IDE) have debugging facilities – use them as well. Use text editors which support grammar checks for the language (e.g. VIM, xemacs, jedit, etc.).

3.1 Basic steps for debugging using `gdb` for C++

```
$ g++ -g hello.cpp
```

Using the `-g` flag gives the same output (i.e. `a.out` or any other user-defined name for the executable using `-o` flag), but gives variable names instead of memory locations.

```
$ gdb a.out
```

This runs a.out in a gdb session.

```
(gdb) r
(gdb) r arg1 arg2
(gdb) r < file1
```

Depending on how the `a.out` is intended to run on the terminal (i.e. without arguments, or with arguments, or with a feed into a file), the `gdb` session can be driven by *running* (`r`) a.out with similar set of arguments.

```
(gdb) help
(gdb) h breakpoints
```

Help topics or content on a specific topic, e.g. breakpoints, can be achieved.

```
(gdb) q
```

Quitting the gdb session.

Some more useful `gdb` commands (`http://www.cs.utexas.edu/users/ygz/378-03S/IBM-debug.pdf`):

- `attach, at` – Attach to an already running process.
- `backtrace, bt` – Print a stack trace.
- `break, b` – Set a breakpoint.
- `clear` – Clear a breakpoint.
- `delete` – Clear a breakpoint by number.
- `detach` – Detach from the currently attached process.
- `display` – Display the value of an expression after execution stops.
- `help` – Display help for gdb commands.
- `jump` – Jump to an address and continue the execution there.
- `list, l` – Lists the 10 lines.
- `next, n` – Step to the next machine language instruction.
- `print, p` – Print the value of an expression.
- `run, r` – Run the current program from the start.
- `set` – Change the value of a variable.

For details on these options on gdb for stepping through your code, use the articles at:

- `https://betterexplained.com/articles/debugging-with-gdb/`

- http://csiflabs.cs.ucdavis.edu/~ssdavis/30/gdb_Tutorial.pdf
- http://heather.cs.ucdavis.edu/~matloff/debug.html is tutorial on `ddd` by Prof. Matloff (University of California at Davis), where `ddd` is a graphical user interface (GUI) to `gdb`. The slide presentation of the tutorial is available at http://heather.cs.ucdavis.edu/~matloff/Debug/Debug.pdf

Overall, useful options in `g++` (from https://www.seas.upenn.edu/cets/answers/gcc.html:

- `-o outputfile` – To specify the name of the output file. The executable will be named `a.out` unless this option is used.
- `-g` – To compile with debugging flags, for use with gdb.
- `-L dir` – To specify directories for the linker to search for the library files.
- `|-l library|` – This specifies a library to link with.
- `|-I dir|` – This specifies a directories for the compile to search for when looking for include files.

4 References

In C, variables are defined using either variable type or pointers to variable types. In C++, *references* are considered to be synonymous to *variables*. A reference to a variable is treated as an *alias*. Both statements can be used:

```
int int_value;
int &ref = int_value;
```

The idea here is that `int_value` and `ref` refer to the same memory location by virtue of their definition. Hence, they can be used interchangeably, for the location or for the variable itself. However, the definition of `ref` referencing to a variable such as `int_value` is needed. Independent of such a referencing, the compiler throws an error, e.g. for the following:

```
int &ref;
```

When the address-of operator `&` is used with a reference, it gives the address of the variable to which the reference applies. However, this is different from pointers, which are variables themselves. The address of a pointer variable, on the other hand, has nothing to do with the address of the variable pointed to.

Different from C, pass by reference functions can be written differently:

```
void modify_p( int *ptr_int )  *ptr_int = 27 ;  // C-style
void modify_r( int &int_value )  int_value = 37 ;  // C++-style
```

Usage in the `main` function is as follows:

```
int main()  int p = 18 ; modify_p(&p) ; modify_r(p) ; return 0 ;
```

We can consider C++ functions to do one of the two tasks:

- modify the input parameter of the function, or,
- inspect the input parameter (usually done by using a copy of the variable or a `const` pointer.

Even though `modify_r(p)` suggests that a copy of `p` is used, in reality, the *reference* to `p` is used and hence, the *variable* itself is modified. Thus, the two tasks can be done using the address-of `&` operator. Hence, in comparison to copy-by-value or the `C`-style, the use of references is recommended owing to its flexibility and genericity. Additionally, passing a reference to an object completely avoids the use of *copy constructors*.

What needs to be noted is that C++ compilers treat references as const pointers. The programmer, on the other hand, does not worry about the levels of indirection. Sometimes indirection can lead to unintended errors, e.g. consider this code:

```
extern int *ip, &ir;
ip = 0 ; // reassigns pointer to a 0-pointer //
ir = 0 ; // the value of the int variable it refers to is now zero.//
```

To drive home the point – pointers can be reassigned to point to different variables, if they are not `const` pointers. However, references *need* to refer to a variable, and cannot be re-aliased to another variable. Thus, they are like `const`-pointers.

5 Example Codes and Practicals

We shall use a few examples to articulate various aspects of C++ programming.

1. Writing a C++ class for `Matrix`:
   - `class-v1`: Setting up set functions; specifying scope (Matrix::) in the definition and implementation.
   - `class-v2`: Setting up friend functions.
2. `references_test.cpp` and `references_functions.cpp` for demonstrating references and the different function types in `C++` with the use of references.
3. Back to `class-v2/Matrix.h`, to show how references are used in class interface.