C Notes #1
Introduction to the C language for Java Programmers

Of all the popular high level programming languages today (C, C++, C#, Java, Basic), C is considered the lowest level of the HLLs. Some authors call C "structured assembler". C supports structured programming, but it predates object-oriented programming (OOP).

In OOP languages like Java and C++, the fundamental program building block is the **class**. In Java, the OO paradigm is taken even further than in C++, to the extent that Java programs at the top level are composed **only** of classes. All functions (methods) belong to some class. To execute a Java program, a class containing a main method must be loaded into the Java Virtual Machine or JVM by the Java interpreter. In C++, classes and member functions can be combined with **global functions**, or functions that **don't belong to any class**. In fact, one global function named "main" must be present in every complete program.

C is similar to C++, but since it predates the development of OOD/OOP ideas, it does not support classes. In other words, the **function** is the basic program building block, and all functions are global. A complete executable C program must contain exactly one function named **main** (no function overloading in C).

Similarities between C, C++, and Java

Most keywords, operators, expression syntax, statement syntax, etc. mean almost the same thing in C, C++, and Java. But on less superficial topics (compilation, memory management, etc.), the languages can be quite different.

CPUs, Instructions, Languages (Machine, Assembly, and High Level)

Every computing platform has at its core a specific kind of hardware chip that defines the computational instructions built into the hardware that are considered the native instructions that the chip can directly execute. This chip is called the CPU (central processing unit). Each kind of CPU is built to recognize a unique set of instruction codes. Collectively these codes are called a language: machine language if in binary form or assembly language if in textual mnemonic form. So the Intel Core recognizes one set of codes, the Sun SPARC another set, the AMD ARM family of chips still another.

Writing programs in assembly is tedious and error prone, and a program painstakingly written for one CPU will not run on a different CPU. To avoid this problem, high-level languages (HLLs) like C, C++, Java, C#, and Python have been developed to allow the programmer to focus on the general logic to solve a problem without committing to the details of a specific chip.

Compilers and Interpreters

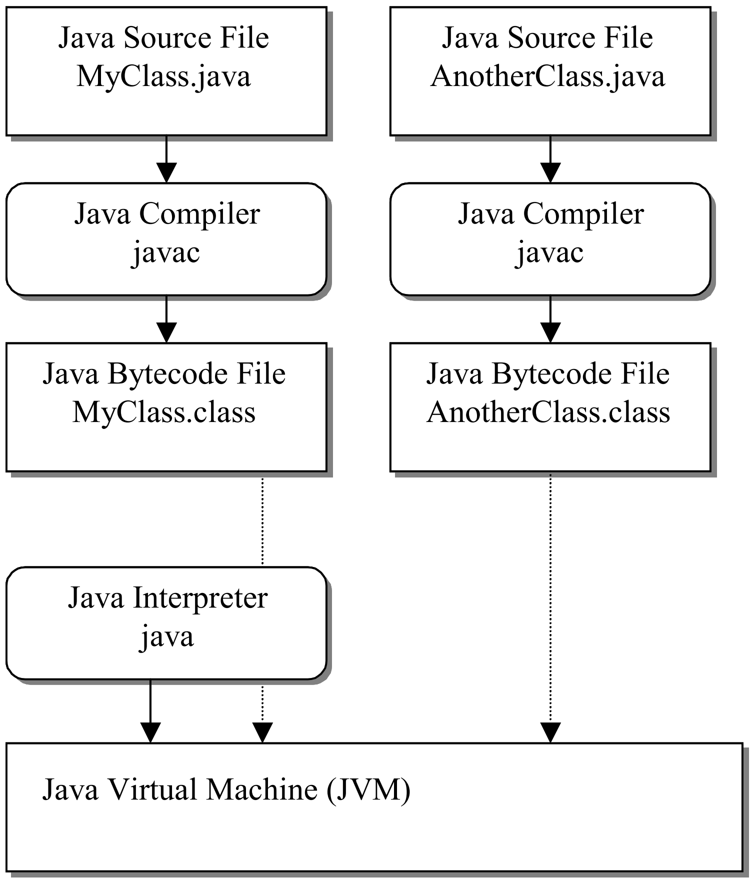
But the problem remains that the CPU is built to execute only its native instructions, not an HLL program. To solve this problem, a series of "helper" or "translator" programs have been developed called compilers, assemblers, and interpreters.

A compiler is a translator that converts the statements in a program from some HLL to the machine language instruction codes for a particular CPU. A compiler works by taking a program written in an HLL, "compiles" or "translates" its statements into the equivalent CPU instruction codes, and deposits these in a new result file, effectively producing a new version of the original program. This new version can then be executed directly by the CPU. Compilers are specific to both the language being compiled and the CPU instruction codes to be produced. For example, a C++ compiler designed for a Pentium-based IBM PC will not run on a PowerPC based Apple Macintosh. As another example, a FORTRAN compiler for an IBM PC cannot compile C programs for an IBM PC.

An interpreter is a program designed for a specific language (the language being interpreted) and customized for a specific CPU. The interpreter takes the program statements written in the language being interpreted and executes them directly, without first translating the program as a whole into machine language. Interpreters work by having at their disposal a set of predefined functions or methods that have been designed and precompiled to mimic the execution of a particular statement from the language being interpreted. When an interpreter interprets a program, the program is read as-is, without first being compiled or assembled. The interpreter reads the statements that make up a program one at a time, and for each statement, looks up and executes the appropriate machine-language procedure for the local CPU that is logically equivalent to the statement being interpreted.

There are trade-offs to consider between interpretation and compilation. Compilers generally produce code that runs faster than an interpreter. But interpreters can reduce the time to develop an initial application prototype, which is useful in cases where the final requirements of an app have not been fully defined, and the developer needs time to experiment with a partially completed app that may still undergo design modifications. Some interpreters even allow a program to be modified interactively during testing.

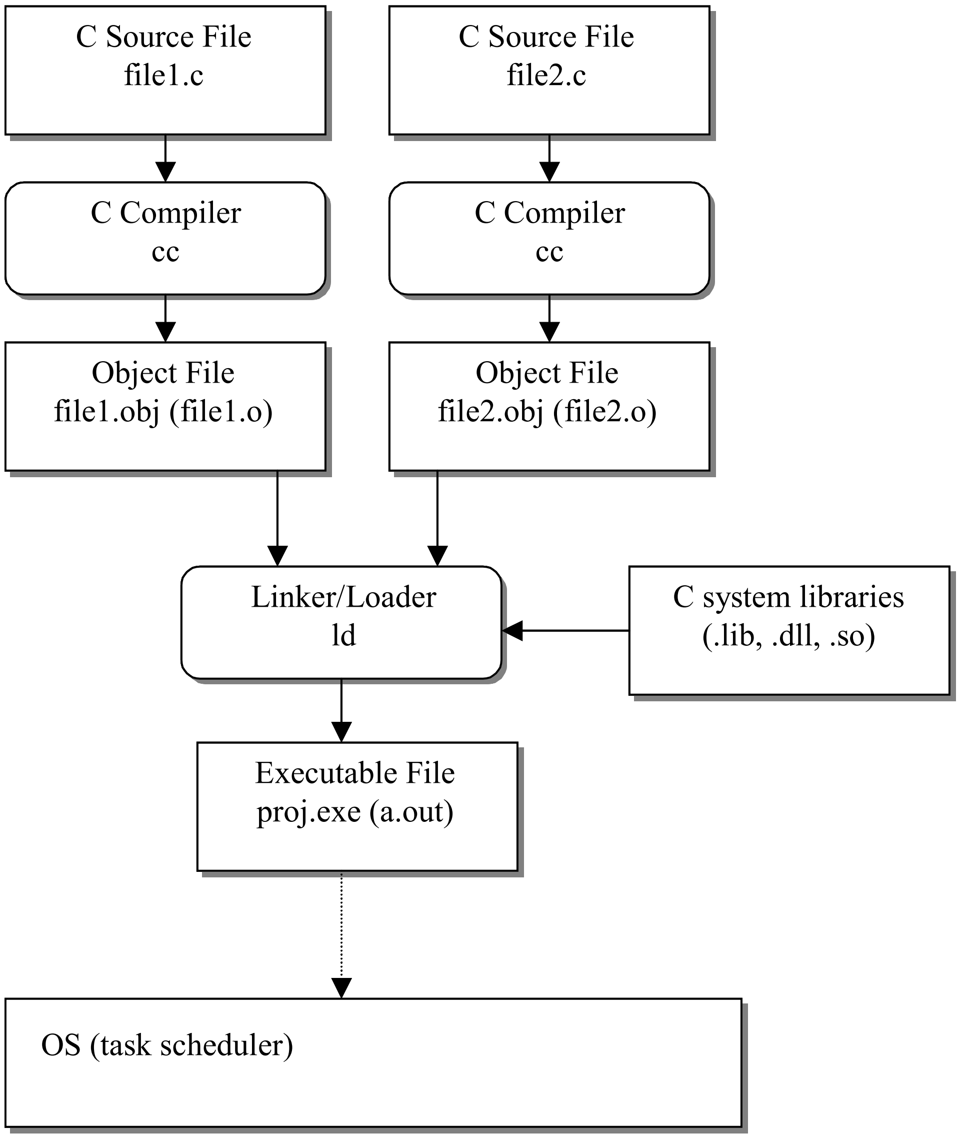
Java is a hybrid language. The most common implementation of Java is to use a pseudo-compiler that translates the statements of the Java source file into a binary instruction format called bytecodes. This is what is inside the class files produced by a Java compiler (class files are the files with filename extension ".class"). Bytecodes do not correspond to the instruction code set for any specific CPU, they are an intermediate machine-independent format. An advantage is that class files compiled on one CPU can be moved to another computer that is based on a different type of CPU and reused without recompilation. In order to execute a bytecode file, there must be available an interpreter called a Java Virtual Machine (JVM) which reads and interprets the bytecodes for a specific CPU. C# and the Microsoft .NET platform use a similar approach.



C and C++ are categorized as fully compiled languages. Source code files in C or C++ are compiled directly into the machine language codes of a specific CPU, rather than machine-independent bytecodes. Object files compiled on one kind of CPU cannot be moved to a computer based on a different kind of CPU and reused. The source file must be moved recompiled on the new computer.

In Java, a program that is ready to be executed by the JVM consists of a collection of class files previously compiled. Unlike for a fully compiled language, these individual class files are never linked together into a single executable file. Instead, when a Java program begins execution, the interpreter (the JVM) is launched, the class file that contains the main method is loaded into the JVM, and the main method is invoked to start the program. As the program runs, any other class files that are needed are dynamically loaded into the JVM when the code that needs them is reached.

In C and C++, the model is different. All the functions and libraries that make up a program can be distributed across multiple source files as the user sees fit and compiled separately. But the individually compiled results are not directly executable. They must first be **linked** together to form a single executable file before it can be submitted to the OS for execution. The executable program is loaded into main memory as a whole. So in addition to a compiler, the development environment for C, C++ and other compiled languages includes another software helper program called a **linker**. The linker is responsible for gluing all the pieces of the program together and making sure nothing is missing (aka address resolution).



Significant Differences between C and C++/Java

Objects

- Since C is not object oriented, concepts such as **public** and **private** are not applicable.
- C does support a user-defined type feature called **struct**, but struct is more limited than a class.
- The keyword **static** is used in C, but it doesn't carry the same meaning as in Java. In Java, **static** refers to a feature that is general to a class rather than an object. In C, **static** generally refers to temporary local variables that would normally appear and disappear as needed as the program executes, but which are instead locked into memory for the entire execution of the program.

Built-in or Primitive Data types

- Similar to C++ and Java
```
int, long, float, double, char, unsigned
```
- Like C++ (and unlike Java), size and range of datatypes can vary from compiler to compiler.
- The char datatype is one byte (8 bits), unlike Java.
- No boolean type in C. Boolean values must be simulated with **int**
  - false represented by 0
  - true represented by any nonzero value, but usually 1.

Strings

- No separate data type in C, strings are not objects in C
- Handled as null-terminated arrays of character constants
```
char c[] = "this is a string";
```
- Large library of String functions
```
#include <string.h>
int x = strlen("this is a string");
char s1[] = "some string";
char s2[80];
strcpy(s2,s1);
```
- In this example, the strcpy() function performs the copy, rather than the assignment operation =

Operators

Like C++ and unlike Java, no distinction between short-circuit and full-circuit logical operators.

&&:	**the AND operation in C**
&:	**the bitwise-AND operation in C**

Pointers and References

Like C++ and unlike Java, C supports pointers.
```
int *p;     // p is a "pointer to int" variable
int x;
p = &x;     // assigns the address or location of x to p
```
Unlike C++, C does not support references.
```
int x;
int &y = x;         // available in C++, not in C
                    // y becomes a reference to x
```
The term **reference** means different things in C++ and Java.

Functions

All functions in a C program are global. Even global data is allowed.
```
int a() { ... }
void b() { ... }
double c() { ... }
int x, y, z;        // vars not declared inside a function
                    // are global
void main() { ... }
```
Classes don't exist so there are no member functions.

Parameter Passing to Functions

Parameter passing in C is strictly by value. But since C supports pointers, by reference parameters can be simulated with pointers.

```
void f1(int *x) { *x = 10; }
void f2()       { int y; f1(&y); }
```

Global vs. Local Variables

A variable that is declared inside the body of a function is called a **local variable**. This means that the usage of the variable is constrained to be local to the function that declares it. Local variables have a storage class called **automatic**. An automatic variable gets created in memory (in a region called the **stack**) whenever its containing function is called. The variable is destroyed (erased from memory) as soon as the function terminates. Technically, local variables can be declared inside of any block starting with curly braces "{", but overuse of this feature can lead to programs that are a little hard to read. A good practice in C is to declare local variables together as a group at the top of the body of the function.

Global variables are different and don't have an exact counterpart in Java. The closest that Java has to global variables are **public static** variables. A **C global variable** is a variable that is declared outside of any function, also called the **top level** of the program. Global variables have a storage class called **static**. The keyword **static** in C is used **differently** from Java. In C, **static** means that a variable is allocated its storage as soon as the program starts to execute, and it remains in memory until the entire program terminates. Global variables in C are inherently **public**. Once a global variable is declared, it can be used by any other function in the program (it's visible and shareable). Global data is an important way for two functions to exchange information, rather than using function calls and return values, but overuse of global variable shares can lead to programs that are hard to debug. In fact the open sharing of all global data in a C program is one of the important motivations behind the development of pu

Example

```
int y;                    // global variable y

void a() {
      y = 15;             // function "a" stores a value into y
      ...
}

void b() {
      int z = y + 3;      // function "b" retrieves a value from y
      ...
}
void main() {
      a();
      b();
      ...
}
```

Function Signatures (prototypes)
Files are compiled by reading from top to bottom. If a function is used before it is declared, there is a potential forward reference problem. Can be fixed by adding a **prototype** before the first usage.

```
double docalc(double,double);            prototype or signature

void main() {
        double x = docalc(5.0, 6.0);     usage before definition
}

double docalc(double x, double y) {      definition
        ...
}
```

Static vs. Dynamic Memory
Like C++ and unlike Java, C supports both static and dynamic allocation of memory for arrays and other extended structures.

```
int x[100];                  // x is a statically allocated array (this is not permitted in Java)

int *x;                      // similar to "int[] x;" in Java
x = (int *) malloc(100*4);   // here, x is dynamically allocated (this is similar to Java)
                             // in Java we would write "x = new int[100];"
...
free(x);                     // dynamically allocated space is being deallocated here
                             // (not necessary in Java where it is automatically handled
                             // by the garbage collector (GC)
```

Also like C++ and unlike Java, C has no garbage collector. The allocation function in C is called “malloc()”. The deallocation function is called “free()”.

```
int *p;
p = (int *) malloc(1000);        // allocates memory by taking it from the “heap” and
                                 // giving it to your program, accessible via p
...
free(p);                         // returns the previously allocated memory to the “heap”
                                 // so that it can be reused later; this step is handled
                                 // automatically in Java by the garbage collector; it is
                                 // not automatic in C and C++
```

Statically allocated arrays (not allowed in Java) do not have to be deallocated. The space for a statically allocated array is automatically released when the function in which it was declared returns or otherwise terminates.

```
void f() {
        int x[100];
        ...
        return;                  // all statically allocated memory used by function f is
                                 // automatically reclaimed here.
}
```

I/O
C has library functions analogous to Java’s System.out.println() and System.in.read(). They are the special formatted I/O functions called “**printf**()” and “**scanf**()”

```
int x;
printf(“Hello, world!”);
printf(“Please enter a value for x:  ”);
scanf(“%d”, &x);
printf(“The value is %d\n”, x);
```

printf and **scanf** use format strings for specifying the format of I/O. scanf requires a **pointer** to a variable where input is to be stored. Not supplying a pointer to scanf is probably the single most common error for new C programmers.

scanf() combines the functions of the Java input and parse operations. There are alternative I/O commands that more closely follow the Java style of receiving input in textual or string form, then parsing it internally to convert it to a different form such as an integer or floating point number.

getline()reads in a line of input in the form of a string
atoi() (stands for “ascii-to-int”) parses a string into an int
atof() (stands for “ascii-to-float”) parses a string into a double (not a float!)

Preprocessor
Like C++ and unlike Java, certain features of the C language are actually supported by a preprocessor, which is not actually part of the language. The #define preprocessor directive is used for both introducing a symbolic alias for a constant value and for parameterized macros. The #include preprocessor directive is used to physically add the content of separate files (usually header files) to the current file during compilation. The #ifdef preprocessor directive can conditionally compile statements, so that one source file can be used to manage multiple versions of an executable. For example one version with debugging output, one with no debugging output.

Tools
Like C++ and unlike Java, C is compiled to machine code. Individual files can be separately compiled to produce relocateable object files. These can be combined in a later step by the linker to produce the final executable or non-relocateable object file.

Editors and Compilers
In the PC labs (EA 1210, 1211), the easiest way to get started with C is to use the **GWD** text editor (shareware, go to www.gwdsoft.com for more info). It has been configured by the IT staff to include menu items for compiling, linking, and executing C/C++ files. A more detailed tutorial with screen captures will be provided later, but if you feel ambitious, try the following:

1. Launch the GWD editor from the Start:Programs menu.
2. Create a new C source file (filename extension should be .c, file name can be anything you want).
3. Type in a simple program like “Hello, world!” (see code below).
4. From the “Tools” menu, select “C compiler”.
5. From the “Tools” menu, select “Execute”.

A more “robust” or “industrial-strength” development environment is the **Microsoft Visual Studio.NET** suite of applications, which is also installed in the PC labs (EA 1210, 1211). This is a powerful IDE (integrated development environment) in which the editor and all program development tools are integrated into one GUI. It’s mostly used for C++ and C#, but it can also be used to edit, compile, and execute C programs. A tutorial will also be provided for this environment. You’re not required to use this application, but it is popular in industry, so it’s a useful one to be familiar with.

Example: the "Hello, world!" program (see K&R, p. 7)

The simplest possible C program that actually does something:

```
#include <stdio.h>

int main(void) {
      printf("Hello, world!\n");
      return 0;
}
```

Here's a pseudo translation of this C program back into Java for your reference:

```
public class Hello {
      public static void main(String[] args) {
            System.out.println("Hello, world!");
      }
}
```

Note that in the C program, the class definition has been "stripped away", leaving only the main function (method). The Java keywords **public static** are not used (for now), so we throw them away. The arguments to the main function (method) are different, but in C you have some choices. The simple console output statement in C is **printf()**, not **System.out.print().**

It's possible to configure your main function in C in a number of ways depending on how your application will interface with the OS console (or parent script). All of the following are possible:

```
      int main(void) { ... return 0; }
      int main(int argc, char *argv[]) { ... return 0; }
      void main(int argc, char *argv[]) { ... }
      void main(void) { ... }
      void main() { ... }
```

Older C compilers will also accept

```
      main() { ... }
```

The general rule is: if you need to access command line arguments, declare the argc/argv params to main, and if the return value from your program will be used by some other application within a "pipeline" of programs that are executed as a group (for example, by a script), then you should give main a return type of int, and actually return a value at the end. By convention, a value of 0 is used to indicate "program terminated normally", while a value other than 0 can be used to indicate some special condition. Return values can be arranged to mean whatever you want them to mean, as long as all programs in the pipeline agree.

Example: read in two ints, add them, print the sum

```
#include <stdio.h>
int main(void) {
      int x, y, z;

      printf("Enter the first integer:  ");
      scanf("%d",&x);
      printf("Enter the second integer:  ");
      scanf("%d",&y);
      z = x + y;
      printf("The result is %d\n", z);

      return 0;
}
```

The most difficult concept to grasp in this example is the use of pointers by scanf. Scanf is a library function that obtains textual input from the keyboard, parses it if necessary, and stores the results in a variable indicated by the pointer argument. In the statement

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

the first argument is a format string. The characters %d inside the string are an indicator to scanf to treat the next input from the keyboard as an integer ("%d" is the scanf code for "integer"). The second argument is &y, which is a pointer to the variable named y. Since scanf is a separate function from the function that it is called from (in this case, it is called from the main function), scanf needs to store the result of the input operation in the variable declared inside another function. The value that must be passed to scanf is therefore the location or address of the variable, **not its current value**. The statement

```
scanf("%d",y);
```

is **incorrect**. Even though it will compile, it will most likely crash. This is because this statement only passes to scanf the old contents of y, information that is completely useless to scanf. Scanf needs an address, a location, so that it can place the result of the input operation into the memory location chosen by the programmer.