Introduction: C

Dr Andrea Bracciali - Dr Deepayan Bhowmik

In this lecture

- The C programming language
- To C from Java

- Hello World

- Code, compile, execute

Learning a new language, C!

- New and different way of doing things
- New and different way of addressing problems and programming.

- Low(er)-level language, most used for system programming, OSs, Networks, …

- … and in many industrial systems and commercial applications

Learning a new language, C!

- Most older code is written in C (or C++)
  - Linux
  - Windows
  - Most Java implementations
  - Most embedded systems

- Philosophical considerations:
  - Being multi-lingual is good!
  - Should be able to trace program from UI to assembly

Language generations and abstraction levels

- Binary, assembly
- Fortran, Cobol
- PL/I, APL, Lisp, … CAML, Haskell…
- C, Pascal, Ada
- C++, Java, Modula3, SmallTalk
- Scripting: Perl, Tcl, Python, Ruby, …
- XML-based languages: CPL, VoiceXML

C history

- C
  - Dennis Ritchie (Bell Labs - early 1970s) in the context of Unix
  - *systems* programming language
    - make OS portable across hardware platforms
    - not necessarily for applications – could be written in Fortran or PL/I
- C++
  - Bjarne Stroustrup (Bell Labs), 1980s
  - object-oriented features
- Java
  - James Gosling in 1990s, originally for embedded systems
  - object-oriented, like C++
  - ideas and some syntax from C

C for Java programmers

- Java is mid-90s high-level OO language
- C is early-70s *procedural* language

- C advantages:
  - Direct access to OS primitives (system calls)
  - Fewer library issues – just execute
- (More) C disadvantages:
  - language is portable, APIs are not
  - memory and “handle” leaks
  - preprocessor can lead to obscure errors

Why learn C (after Java)?

- Both high-level and low-level language
  - OS: user interface to kernel to device driver
- Better control of low-level mechanisms
  - memory allocation, specific memory locations
- Performance *sometimes* better than Java
  - usually more predictable (also: C vs. C++)
- Java hides many details needed for writing OS code

  But C comes with…
  - Memory management responsibility
  - Explicit initialization and error detection
  - Generally, more lines for same functionality
  - More room for mistakes

C vs. Java

Java	C
object-oriented	function-oriented
automatic memory management	function calls (C++ has some support)
no “explicit” pointers	pointers (memory addresses) common
by-value	by-value/ by ref parameters*
exceptions, exception handling	if (f() < 0) {error} OS signals

C vs. Java

Java	**C**
length of array	on your own
string as type	just bytes (char []), with 0 end
dozens of common libraries	OS-defined

C vs. Java

- Java program
  - collection of classes
  - class containing main method is starting class
  - running `java` *`StartClass`* invokes *`StartClass`*`.main` method
  - JVM is the execution environment ("same" for each platform)
  - JVM loads other classes as required

- C
  - collection of functions
  - one function – `main()` – is starting function
  - running executable (default name a.out) starts main function
  - typically, single program with all user code linked in – but can be dynamic libraries (.dll, .so)
  - platform-specific

Simple example – Hello World

```
#include <stdio.h>

int main(void)
{
    /* print out a message */
    printf(“Hello World. \n \t and you ! \n ”);

    return 0;
}
```

Simple example – Hello World

```
#include <stdio.h>

int main(void)
{
    /* print out a message */
    printf(“Hello World. \n \t and you ! \n ”);

    return 0;
}
```

- `#include <stdio.h>`
  - include header file `stdio.h`
  - # lines processed by *pre-processor* - no semicolon at end
  - Lower-case letters only – C is case-sensitive

Simple example

```
#include <stdio.h>

int main(void)
{
    /* print out a message */
    printf(“Hello World. \n \t and you ! \n ”);

    return 0;
}
```

- `int main(void){ … }`
  - it is a function ! the body of the program
  - it returns an integer – a termination code
  - Is the only code executed

Simple example

```
#include <stdio.h>

int main(void)
{
    /* print out a message */
    printf(“Hello World. \n \t and you ! \n ”);

    return 0;
}
```

- `printf(“Hello World. \n \t and you ! \n”);`
  - prints a desired message “Hello World”
  - `\n` = newline, `\t` = tab
  - \ in front of other special characters within `printf`.

```
printf(“Have you heard of \”The Rock\”? \n”);
```

Simple example

```
#include <stdio.h>

int main(void)
{
    /* print out a message */
    printf(“Hello World. \n \t and you ! \n ”);

    return 0;
}
```

- `return 0;`
  - terminates the function (and returns control)
  - here 0 means “no problems” – 0 also stands for FALSE

Simple example

```
#include <stdio.h>

int main(void)
{
    /* print out a message */
    printf(“Hello World. \n \t and you ! \n ”);

    return 0;
}
```

```
$Hello World.
         and you !
$
```

Comments

- *`/* any text until */`*
- `//` C++-style comments – can be used most of the times!
- Convention for longer comments (be consistent!):

```
/*
 * AverageGrade()
 * Given an array of grades, compute the average.
 */
```

C vs. Java

```
public class hello
{
     public static void main
   (String args []) {
          System.out.println
          (“Hello world”);
     }
}
```

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  puts(“Hello World”);
  return 0;
}
```

What does this C program do ?

```
#include <stdio.h>

struct list{int data; struct list *next};
struct list *start, *end;

void add(struct list *head, struct list *list, int data};
int delete(struct list *head, struct list *tail);

int main(void)
{
 start=end=NULL;
 add(start, end, 2); add(start, end, 3);
 printf(“First element: %d”, delete(start, end));
 return 0;
}
```

What does this C program do – cont.d?

```
void add(struct list *head, struct list *tail, int data}
{
 if(tail==NULL){
  head=tail=malloc(sizeof(struct list));
  head->data=data; head->next=NULL;
 }
 else{
  tail->next= malloc(sizeof(struct list));
  tail=tail->next; tail->data=data; tail->next=NULL;
 }
}


void delete (struct list *head, struct list *tail)
{
 struct list *temp;
 if(head==tail){
  free(head); head=tail=NULL;
 }
 else{
  temp=head->next; free(head); head=temp;
 }
}
```

Just a taster !

will be clear later on

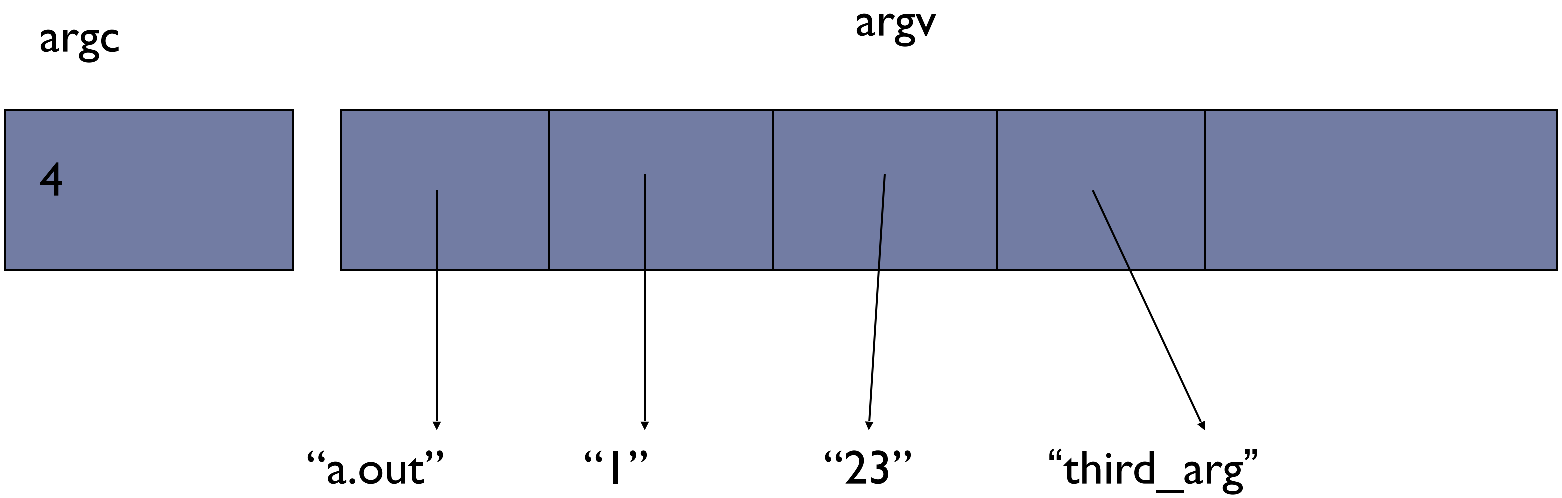
Executing the C program

```
int main(int argc, char argv[])
```

- argc is the argument count
- argv is the argument vector
  - array of strings with command-line arguments
- the `int` value is the return value
  - convention: 0 means success, > 0 some error
  - can also declare as void (no return value)

Executing a C program

- Name of executable + space-separated arguments
- $ a.out 1 23 third_arg



Executing a C program

- If no arguments, simplify:

```
int main(void) {
  puts(“Hello World”);
  exit(0);
}
```

- Note `exit()` instead of return – similar effect.

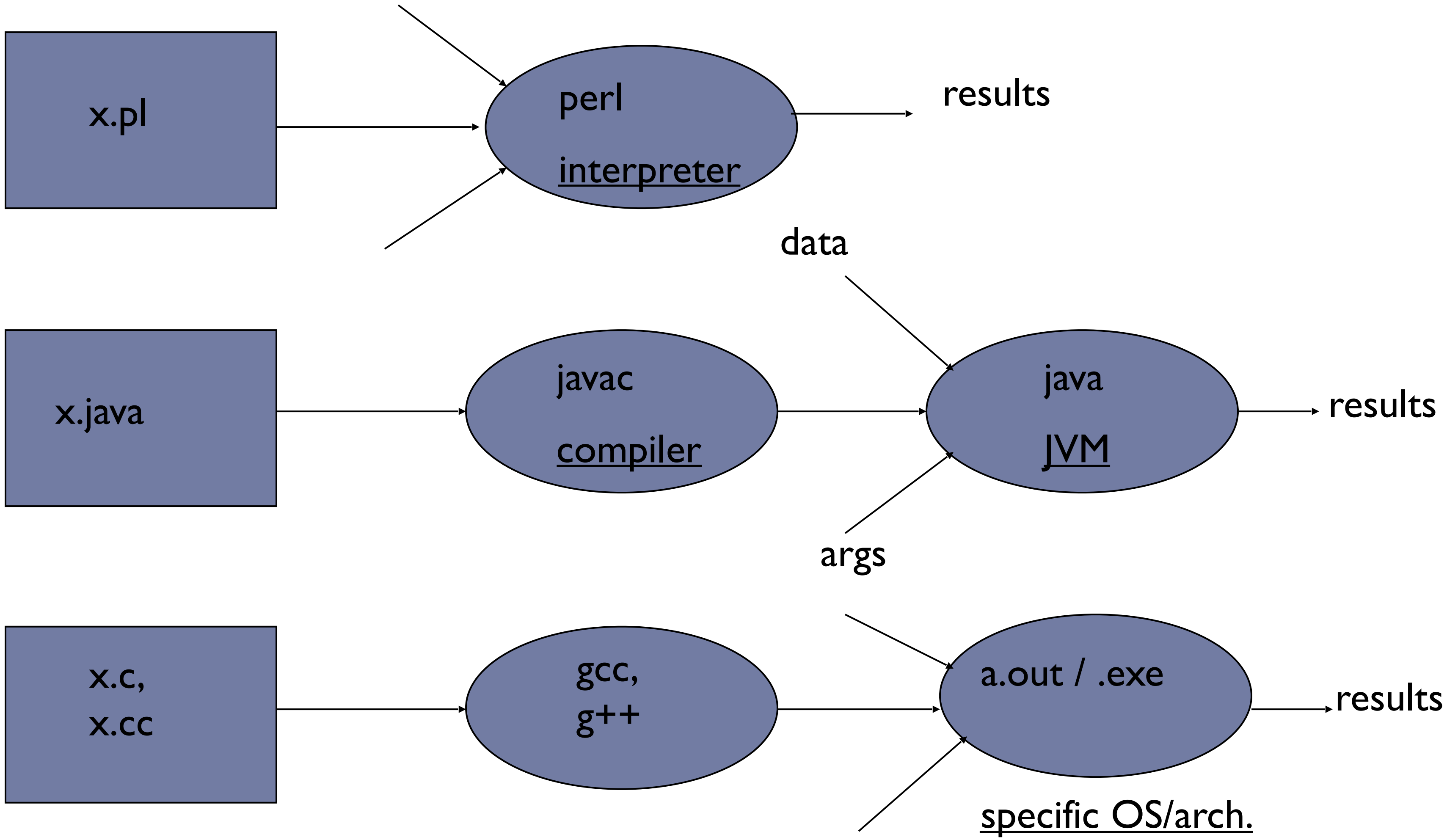
Executing C programs

- Scripting languages are usually interpreted
  - perl (python, Tcl) reads script, and executes it
  - sometimes, just-in-time compilation – invisible to user

- Java programs semi-interpreted:
  - javac converts `foo.java` into `foo.class`
  - not machine-specific
  - *byte codes* are then interpreted by JVM

- C programs are normally compiled and linked:
  - gcc converts `foo.c` into `a.out`
  - `a.out` or `a.exe` is executed by OS and hardware

Executing C programs



The C compiler gcc

- gcc translates C program into executable for some target
- gcc: GNU Compiler Collection, includes
  - Pre-processor (cpp),
  - Compiler (cc1),
  - Linker (ld).

- default file name a.out
- also "cross-compilation" (for another architecture)

```
$ gcc hello.c
$ a.exe
Hello, World!
```

gcc

- Behavior controlled by command-line switches:

-o file	output file name for object or executable
-Wall	all warnings – use always!
-c	compile single module (non-main)
-g	insert debugging code (gdb)
-p	insert profiling code
-l	library
-E	preprocessor output only

Using gcc

- Two-stage compilation
  - pre-process & compile: `gcc –c hello.c`
  - link: `gcc –o hello hello.o`
- Linking several modules:

`gcc –c a.c` → a.o

`gcc –c b.c` → b.o

`gcc –o hello a.o b.o`

- Using math library
  - `gcc –o calc calc.c -lm`

Error reporting in gcc

- Multiple sources
  - preprocessor: missing include files
  - parser: syntax errors
  - linker: missing libraries ←
  - assembler: rare

Error reporting in gcc

- If `gcc` gets confused, hundreds of messages
  - fix first, and then retry – ignore the rest

- `gcc` will produce an executable with warnings
  - don't ignore warnings – compiler choice is often not what you had in mind

- Does not flag common errors
  - `if (x = 0)` vs. `if (x == 0)`

C preprocessor

- The C preprocessor is a macro-processor that
  - manages a collection of macro definitions
  - reads a C program and transforms it
  - Example:

```
#define MAXVALUE 100
#define check(x) ((x) < MAXVALUE)
if check(i) { …}
```

becomes

```
if ((i) < 100)  {…}
```

Advice on preprocessor

- Limit use as much as possible
  - subtle errors
  - not visible in debugging
  - code hard to read
- much of it is historical baggage
- there are better alternatives for almost everything:
  - #define INT16 -> type definitions
  - #define MAXLEN -> const
  - #define max(a,b) -> regular functions
- limit to .h files, to isolate OS & machine-specific code

Too much? Not to worry, we'll get there in time!