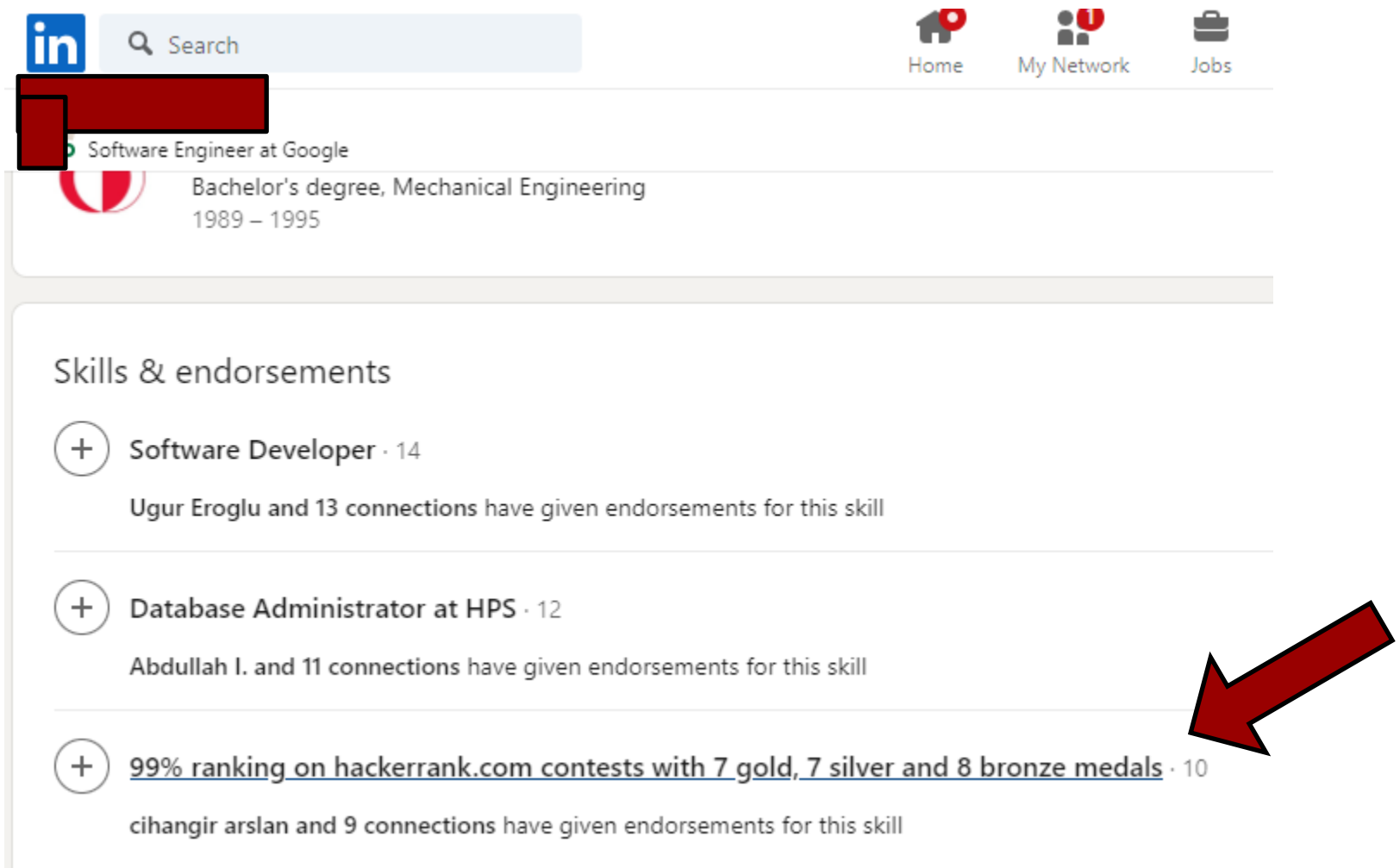
Introduction to C Programming

Agenda for the week

- **Module 2 is available**
  - Lecture 1: Introduction to C Programming
  - Lecture 2: Arrays and Pointers
  - Lab 2: C programming (Multi File)
  - HW3 and HW4: C practice On HackerRank
  - Ungraded Assignments: Kahoot questions are also test review questions.

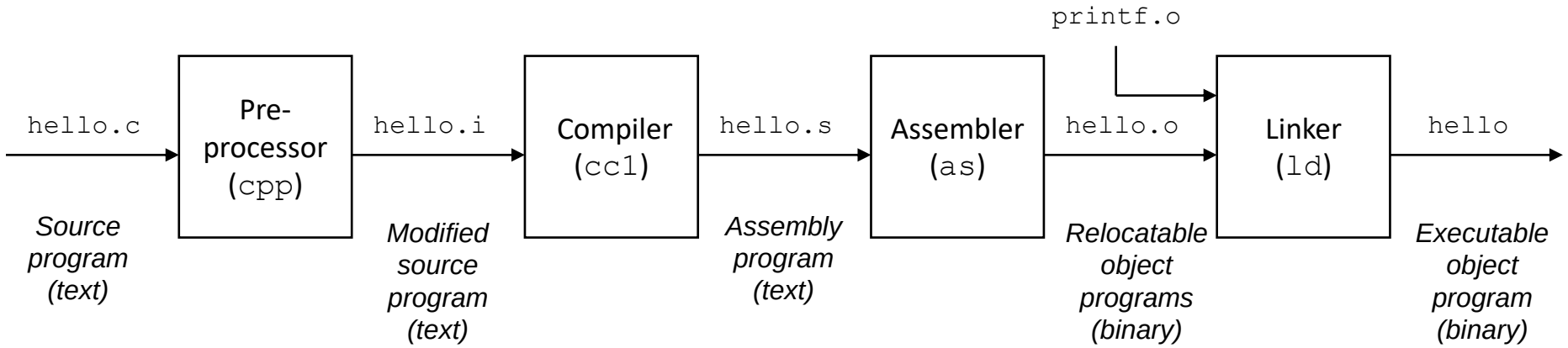
How to get a high paying job?



Outline

- Compilation Process
- JAVA vs C
- Primitive types in C
- C Program Layout
- Basic Data Structures
- Functions
- Compiling

Compilation Process




```
1   #include <stdio.h>
2
3   int main()
4   {
5       printf("hello, world\n");
6   }
```

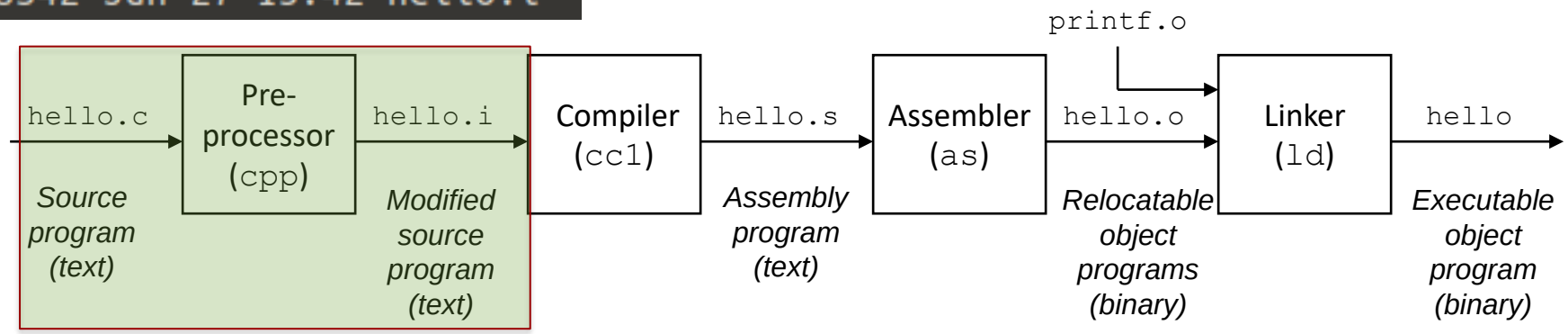
- **Compilation process is composed of**
  - Preprocessor
  - Compiler
  - Assembler
  - Linker

Compilation Process

- **Preprocessing Phase:**
  - The preprocessor **(cpp)** modifies the original program in C according to the directives that begin with # character
  - For example the #include <stdio.h> command tells the preprocessor to read the contents of the system header file stdio.h and insert it directly in the program text.
    - Result is another C program typically with the .i suffix
    - To test use: **gcc –E helloWorld.c –o helloWorld.i**
    - **or : cpp hello.c -o hello.i**

More in Module 5

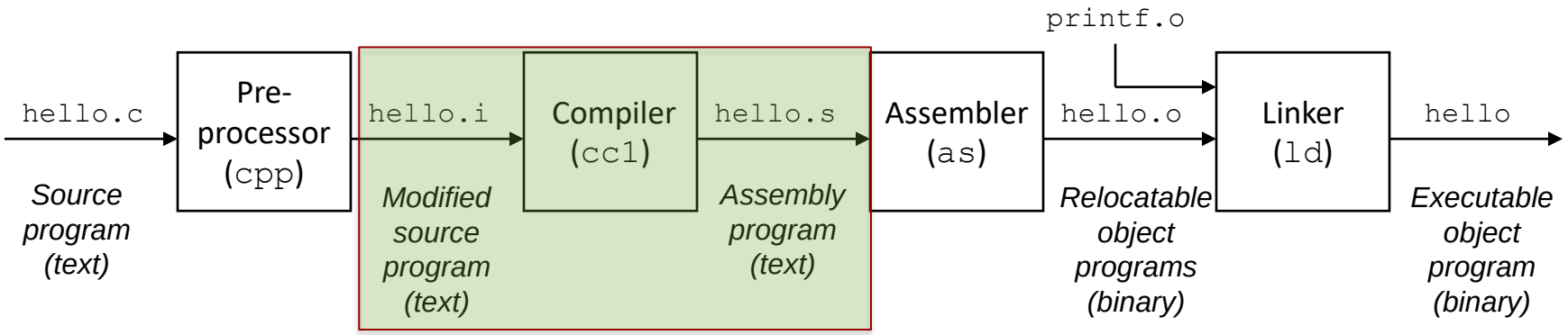
```
   81 Jun 27 12:47 hello.c
18342 Jun 27 15:42 hello.i
```



Compilation Process

- **Compilation Phase:**
  - The compiler (cc1) translates the text file hello.i into the text file hello.s
    - **hello.s contains the assembly-language program.**
    - **This program includes low level machine instructions in a textual form**
    - **Different high level programming languages generate similar assembly programs.**
    - **gcc -S hello.c -o hello.s**

```
main:
.LFB0:
        .cfi_startproc
        leal    4(%esp), %ecx
        .cfi_def_cfa 1, 0
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        .cfi_escape 0x10,0x5,0x2,0x75,0
        movl    %esp, %ebp
        pushl   %ebx
        pushl   %ecx
        .cfi_escape 0xf,0x3,0x75,0x78,0x6
        .cfi_escape 0x10,0x3,0x2,0x75,0x7c
```

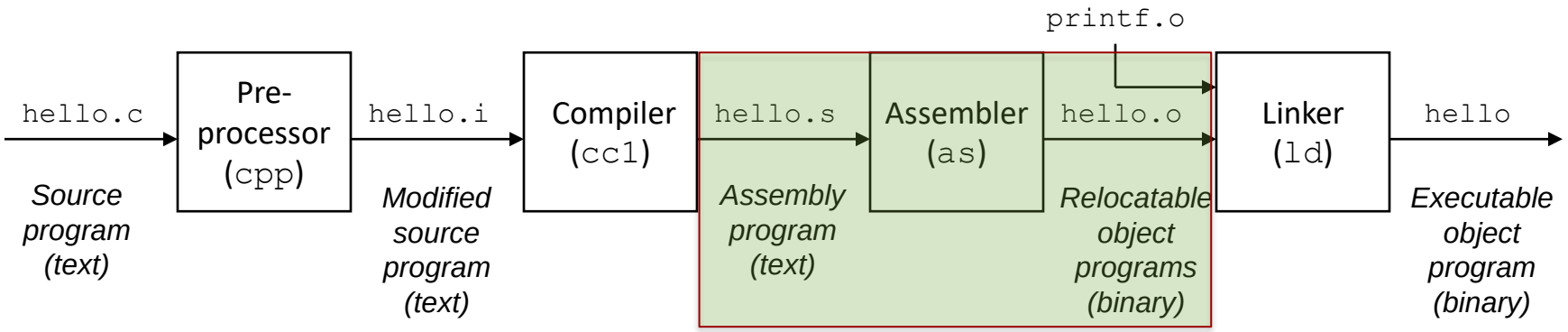


Compilation Process

- **Assembly Phase:**
  - **Assembler translates hello.s into machine language instructions**
  - **Packages them in a form known as relocatable object program and stores the result as hello.o**
  - **Use as: gcc -c hello.c -o hello.o**

```
 GNU nano 2.9.3                                  hello.o

^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^A^@^C^@^A^@^@^@^@^@^@^@^@^@^@^@^
^@^@^@^@^@^@^@^@^@^@^@^@^@^C^@^H^@^@^@^@^@^@^@^@^@^@^@^@^@^C^@^
^M^@^@^^^@^@^@  ^E^@^@&^@^@^@^D^N^@^@ ^@^@^@^B^B^@^@T^@^@^@^B^F
^@^@^@^D^@^@^@^H^@^@^@^A^@^@^@^B^@^@^@^@^@^@^@^@^@^@^@^@^@^X^A^@^@
```

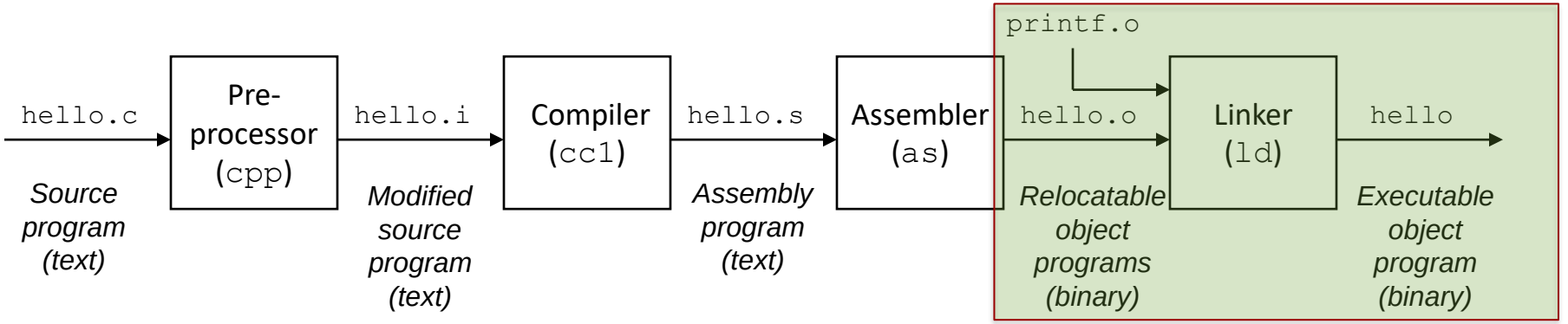
No more human readable!



Compilation Process

- **Linking Phase:**
  - **Hello program calls printf function, which is a part of standard C library. printf is a separate precompiled object: printf.o**
  - **We need to merge printf.o and hello.o**
  - **Linker handles that step**
  - **Result is the hello file**
  - **gcc hello.c -o hello**

More in Module 5



If you understand Compilation System better

- **You can optimize Program Performance**
  - Modern compilers already have optimizers, but you can do fine tuning
- **Understanding link-time errors**
  - What does it mean when a linker reports that it cannot resolve a reference?
  - Why do some linker related errors do not appear until run time ?
  - What is the difference between static library and dynamic library?
  - What happens if you define two global variables in different C files with the same name.
- **Avoiding security holes**
  - Buffer overflow vulnerability
  - Format string vulnerability
  - If you can understand how the program information is stored in memory you can prevent these.

Outline

- Compilation Process
- JAVA vs C
- Primitive types in C
- C Program Layout
- Basic Data Structures
- Functions
- Compiling

Java vs. C

Language Feature	Similar or Different
Control Structures (if, while, for, switch)	Similar - In C you can not use Strings in the expression of a switch - In C 89, you need to declare control variable before the loop
Primitive datatypes(int, char, Boolean, long)	Similar. - Sizes might differ. - No boolean in C (after C99 there is bool under stdbool.h)
Operators (+, -, *, …)	Similar
Casting	Different (Similar Syntax) - Java enforces type safety, no error checking in C.
Arrays	Different (Similar Syntax) -No bounds checking in C
Dynamic Memory Management	Different -allocation: “malloc” in C, “new” in Java -deallocation: “free” in C, automatic in Java

More at: https://introcs.cs.princeton.edu/java/faq/c2java.html

Java vs. C

- **Things that are the same as Java**
  - syntax for statements, control structures, function calls
  - types: int, double, char, long, float
  - type-casting syntax: float x = (float) 5 / 3;
  - expressions, operators, precedence

    ++ -- * / + - % = += -= *= /= %= < <= == != > >= && || !
  - scope (local scope is within a set of { } braces)
  - comments: /* comment */ or // comment

Similar to Java…

- **variables**
  - Must declare at the start of a function or block *(changed in C99)*
  - Don't need be initialized before use *(gcc -Wall* will warn*)*

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int x, y = 5;   // note x is uninitialized(!)
  int k;
  long z = x+y;
  printf("x is '%d'\n", x);
  printf("k is '%d'\n", k);
  printf("z is '%ld'\n", z); // w
  {
    int y = 10;
    printf("y is '%d'\n", y);
  }
  int w = 20;  // ok in c99
  printf("y is '%d', w is '%d'\n", y, w);
  return 0;
}
```

```
cs257@cs257-VirtualBox:~/Desktop$ ./varscope
x is '4445739'
k is '-1075120236'
z is '4445744'
y is '10'
y is '5', w is '20'
```

Similar to Java…

- printf()
- Similar to System.out.printf() in Java

```
int a = 5;
double pi = 3.14159;
char s[] = "I am a string!";
printf("a = %d, pi = %f, s = %s\n",a, pi, s);
```

- Substitutes values for %d, %f, %s etc.
- %d : int, %f : float, %lf : double, %s : string
- \n : new line

  - Look here:
    - http://man7.org/linux/man-pages/man3/printf.3.html
    - Or type man 3 printf

Different! Getting user input

- **Using** scanf **to get user input**
  - Read formatted input from stdin (standard input stream)
  - To read and integer assuming number is an integer:

    scanf("%d",&number)
  - % is format specifier, cannot explain why we are using & now, we need to learn pointers first
  - Rule: & is needed for reading char, int, double but not string
  - Returns an int value
    - If successful returns the number of items read
    - If input is unavailable, special EOF (“end of file”) value is returned
  - Look here:
    - http://man7.org/linux/man-pages/man3/scanf.3.html
    - Or type man scanf

Similar to Java…

- **const**
  - a qualifier that indicates the variable's value cannot change
  - compiler will issue an error if you try to violate this
  - (in Java you use static final double …)

```c
#include <stdio.h>

int main(int argc, char **argv) {
  const double MAX_GPA = 4.0;

  printf("MAX_GPA: %g\n", MAX_GPA);
  MAX_GPA = 5.0;  // illegal!
  return 0;
}
```

Similar to Java…

- **for loops**
  - can't declare variables in the loop header *(changed in c99), otherwise loops are very similar.*
- **if/else, while, and do/while loops**
  - no boolean type *(changed in c99)*
    - *You can now #include <stdbool.h> and use bool type*
  - any type can be used; 0 means **false,** everything else **true**

```
int i;

for (i=0; i<100; i++) {
  if (i % 10 == 0) {
    printf("i: %d\n", i);
  }
}
```

Outline

- Compilation Process
- JAVA vs C
  - Primitive types in C
- C Program Layout
- Basic Data Structures
- Functions
- Compiling

Primitive Types in C

- **Size of each type may change depending on the machine**
- **Integer types**
  - `char`, `int`
- **Floating point**
  - `float`, `double`
- **Modifiers**
  - `short` [int]
  - `long` [int, double]
  - `signed` [char, int]
  - `unsigned` [char, int]

C Data Type	32-bit	64-bit	printf
char	1	1	%c
short int	2	2	%hd
unsigned short int	2	2	%hu
int	4	4	%d / %i
unsigned int	4	4	%u
long int	4	8	%ld
long long int	8	8	%lld
float	4	4	%f
double	8	8	%lf
long double	12	16	%Lf
pointer	4	8	%p

C99 Extended Integer Types

- Solves the conundrum of "how big is an `long int`?"
- Use to make your code portable between 32bit and 64bit systems. **Important for system programming!**

```
#include <stdint.h>

void foo(void) {
  int8_t  a;  // exactly 8 bits, signed
  int16_t b;  // exactly 16 bits, signed
  int32_t c;  // exactly 32 bits, signed
  int64_t d;  // exactly 64 bits, signed
  uint8_t w;  // exactly 8 bits, unsigned
  ...
}
```

```
void sum (int x, int y) {
```

```
void sum (int32_t x, int32_t y) {
```

Outline

- Compilation Process
- JAVA vs C
- Primitive types in C
- C Program Layout
- Basic Data Structures
- Functions
- Compiling

Generic C Program Layout

```c
#include <system_files>
#include "local_files"

#define macro_name macro_expr

/* declare functions */
/* declare external variables & structs */

int main(int argc, char* argv[]) {
  /* the innards */
}

/* define other functions */
```

C Syntax: `main`

- **To get command-line arguments in `main`, use:**

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

- **What does this mean?**
  - `argc` contains the number of strings on the command line (the executable name counts as one, plus one for each argument).
  - `argv` is an array containing *pointers* to the arguments as strings (more on pointers later)

- **Example:**

```
$ ./foo hello 87
```

  - `argc = 3`
  - `argv[0]="./foo", argv[1]="hello", argv[2]="87"`

Simple program

- There are more similarities and differences, first let's start with our first program
- The "hello, world!" program:

```
#include <stdio.h>
int main(void)// main gets no arguments
{
  printf("hello, world!\n");
  return 0;
}
```

Getting Started

- **Make this into a file called hello.c using a text editor**
  - vi, nano, (available on server)
  - geany, gedit, nedit, pico (other simple editors, but not available on server or available through Xserver)

- **Compile into a program and run**

```
$gcc hello.c -o hello
$./hello
hello, world!
```

Outline

- Compilation Process
- JAVA vs C
- Primitive types in C
- C Program Layout
- Basic Data Structures (Arrays)
- Functions
- Compiling

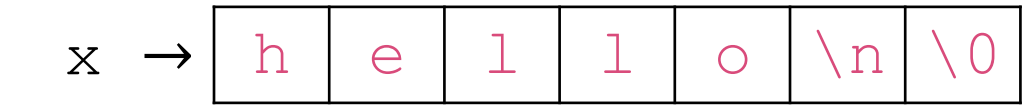
Basic Data Structures

- **C does not support objects!!!**

- **Arrays are contiguous chunks of memory**
  - Arrays have no methods and do not know their own length
  - Can easily run off ends of arrays in C – security bugs!!!

- **Strings are null-terminated char arrays**
  - Strings have no methods, but `string.h` has helpful utilities

```
char* x = "hello\n";
```



- **Structs are the most object-like feature, but are just collections of fields – no "methods" or functions**
- **We cover these futures in more detail**

Data types

- Strings: arrays of type char

```
char string1[7] = "Hello!";
char string2[]  = "Hello!";
char* string3  = "Hello!";
```

- Much more on strings, pointers and arrays later
- Other types: structs, pointers
- Where it is saved is different for the string3. We will learn more later on.

Outline

- Compilation Process
- JAVA vs C
- Primitive types in C
- C Program Layout
- Basic Data Structures
- Functions
- Compiling
- Multi-File C Programs

Function Definitions

- **Generic format:**

```
returnType fname(type param1, …, type paramN) {
                    // statements
                      }
```

```
// sum of integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;

  for (i = 1; i <= max; i++) {
    sum += i;
  }

  return sum;
}
```

Function Ordering

- **You *shouldn't* call a function that hasn't been declared yet**
- **Compiler will assume it returns an int which may not be true (called implicit declaration)**

```c
#include <stdio.h>

int main(int argc, char** argv) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return 0;
}

// sum of integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;

  for (i = 1; i <= max; i++) {
    sum += i;
  }
  return sum;
}
```

You can

- Move sumTo before main
- Declare sumTo before main
- Include a header file which has declaration

Solution 1: Reverse Ordering

- **Simple solution; however, imposes ordering restriction on writing functions (who-calls-what?)**

```c
#include <stdio.h>

// sum of integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;

  for (i = 1; i <= max; i++) {
    sum += i;
  }
  return sum;
}

int main(int argc, char** argv) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return 0;
}
```

Solution 2: Function Declaration

- **Teaches the compiler arguments and return types; function definitions can then be in a logical order**

```c
#include <stdio.h>

int sumTo(int);  // func prototype

int main(int argc, char** argv) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return 0;
}

// sum of integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;
  for (i = 1; i <= max; i++) {
    sum += i;
  }
  return sum;
}
```

To run this code in online debugger:

https://onlinegdb.com/r1Mcq3JZI

Solution 3: Header file

- **Replaces the `#include "sumTo.h"` with the contents of sumTo.h This will effectively work as solution 2.**

```c
#include <stdio.h>
#include "sumTo.h"

int main(int argc, char** argv) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return 0;
}

// sum of integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;
  for (i = 1; i <= max; i++) {
    sum += i;
  }
  return sum;
}
```

sumTo.h

```c
int sumTo(int);  // func prototype
```

Function Declaration vs. Definition

- **C/C++ make a careful distinction between these two**

- **Definition: the thing itself**
  - *e.g.* code for function, variable definition that creates storage
  - Must be **exactly one** definition of each thing (no duplicates)

- **Declaration: description of a thing**
  - *e.g.* function prototype, external variable declaration
    - Often in header files and incorporated via `#include`
    - Should also `#include` declaration in the file with the actual definition to check for consistency
  - Needs to appear in **all files** that use that thing
    - Should appear before first use

Outline

- Compilation Process
- JAVA vs C
- Primitive types in C
- C Program Layout
- Basic Data Structures
- Functions
- Compiling

Compiling

- Difference between compiling w/wo -Wall

- With –Wall you will see warnings

```
cs257@cs257-VirtualBox:~/Desktop$ gcc -Wall varscope.c -o varscope
varscope.c: In function ‘main’:
varscope.c:6:8: warning: ‘x’ is used uninitialized in this function [-Wuninitial
ized]
   long z = x+y;
        ^
varscope.c:8:3: warning: ‘k’ is used uninitialized in this function [-Wuninitial
ized]
   printf("k is '%d'\n", k);
   ^~~~~~~~~~~~~~~~~~~~~~~~
```

- Without wall: No warnings!

```
cs257@cs257-VirtualBox:~/Desktop$ gcc varscope.c -o varscope
cs257@cs257-VirtualBox:~/Desktop$
```

Multi-file C Programs

C source file 1 (sumstore.c)

```c
void sumstore(int x, int y, int* dest) {
  *dest = x + y;
}
```

C source file 2 (sumnum.c)

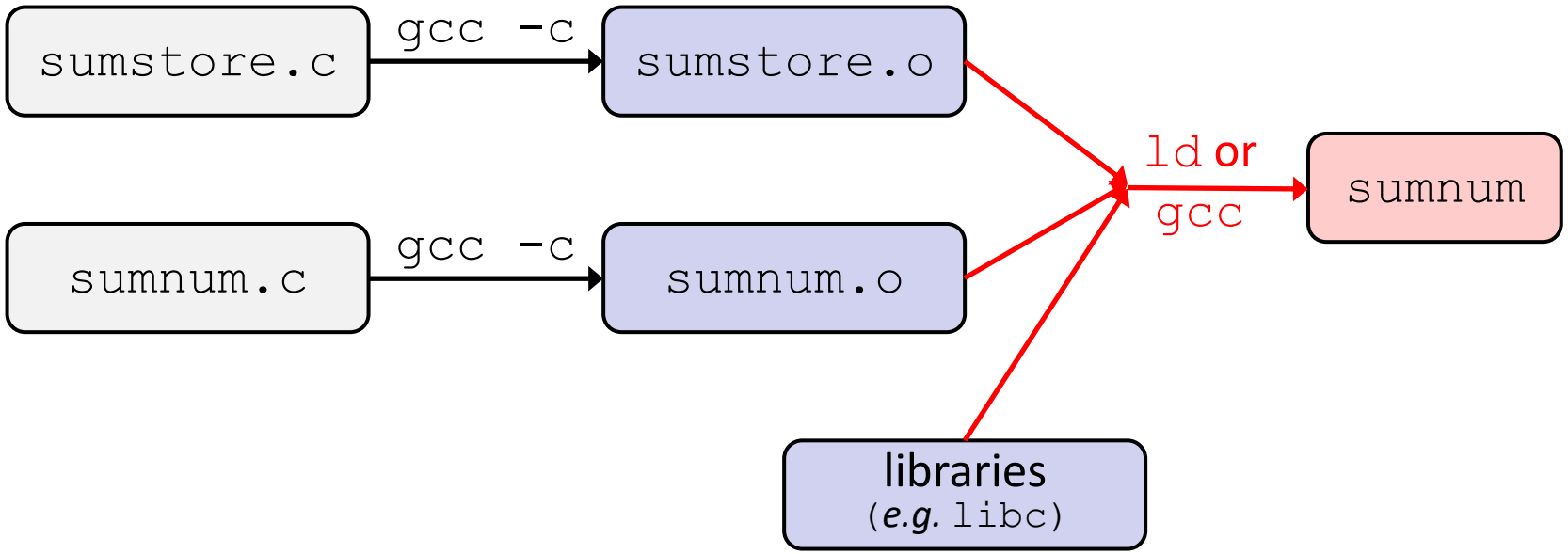
```c
#include <stdio.h>

void sumstore(int x, int y, int* dest);

int main(int argc, char** argv) {
  int z, x = 351, y = 333;
  sumstore(x, y, &z);
  printf("%d + %d = %d\n", x, y, z);
  return 0;
}
```

Compile together:

```
$ gcc sumnum.c sumstore.c -o sumnum
```

Multi-file C Programs

- **The linker combines multiple object files plus statically-linked libraries to produce an executable**
  - Includes many standard libraries (*e.g.* `libc`)
    - A *library* is just a pre-assembled collection of `.o` files



Resources For Practice

- https://www.learn-c.org/
- https://www.w3resource.com/c-programming-exercises/