Chapter 2: Introduction to C

252-0061-00 V Systems Programming and Computer Architecture

Goal

- Introduction to C
  - Enough to program assignments
  - Background for lectures
- Assume you know Java (or C#)
  - E.g. from Parallel Programming
- Non-goal (for now…):
  - Teach details and strict definition of C
  - Teach advanced features/idioms/techniques in C
  - But still much more than you saw last year…

2.1: History and toolchain

Systems Programming and Computer Architecture

History

- Developed 1969-1972 by Dennis Ritchie
  - CPL $\rightarrow$ BCPL $\rightarrow$ B $\rightarrow$ C
  - Highly influenced by DEC PDP-11 architecture
  - Portable across many architectures
- Standards:
  - K&R C (standard was the compiler source!)
  - ANSI C
  - **C99 (we'll use this)**
  - C11 (more recent, less used)
  - C17 (even more recent, bug fixes to C11…)
  - … and many C-like variants

1941-2011

Compared to Java, C#, PHP, Python, etc…

- Very fast
  - Almost impossible to write assembly as fast as a good C compiler
  - Pretty much impossible to compile Java to run as fast as C
- Powerful macro pre-processor (cpp)
- Close to the metal: you can *know* what the code is doing to the hardware

⇒ Remains the language of choice for

- Operating System developers
- Embedded systems
- People who *really* care about speed
- Authors of security exploits

Just some of what you don’t get

- No objects, classes, traits, features, methods, or interfaces
  - Only functions/procedures
  - We will see function pointers later…
- No fancy built-in types
  - Mostly just what the hardware provides
  - Type constructors to build structured types
- No exceptions
  - convention is to use integer *return codes*

Most important difference

- No automatic memory management
  - Lots of things on the stack
  - No garbage collection
  - Heap structures explicitly created and freed
- Pointers: direct access to memory addresses
  - Weakly typed by what they point to

C is about directly building and manipulating structures in main memory!

Syntax: the good news

- Java, JavaScript, C++, and C# syntax almost entirely lifted from C
- Comments (/*...*/, //) the same
- Identifiers same as in Java
  - C# allows more characters in identifiers
- Block structure using { … }

Many other constructs the same or similar.

Syntax: main differences

- List of reserved words is different
- C is run through a *macro preprocessor*
  - String and file substitution
  - Conditional compilation
  - Although C# has preprocessor directives, it does not have a separate preprocessor.
    Moreover there are no macros.

Source code: chapter 2 → `hello`

Hello world

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  printf("hello, world\n");
  return 0;
}
```

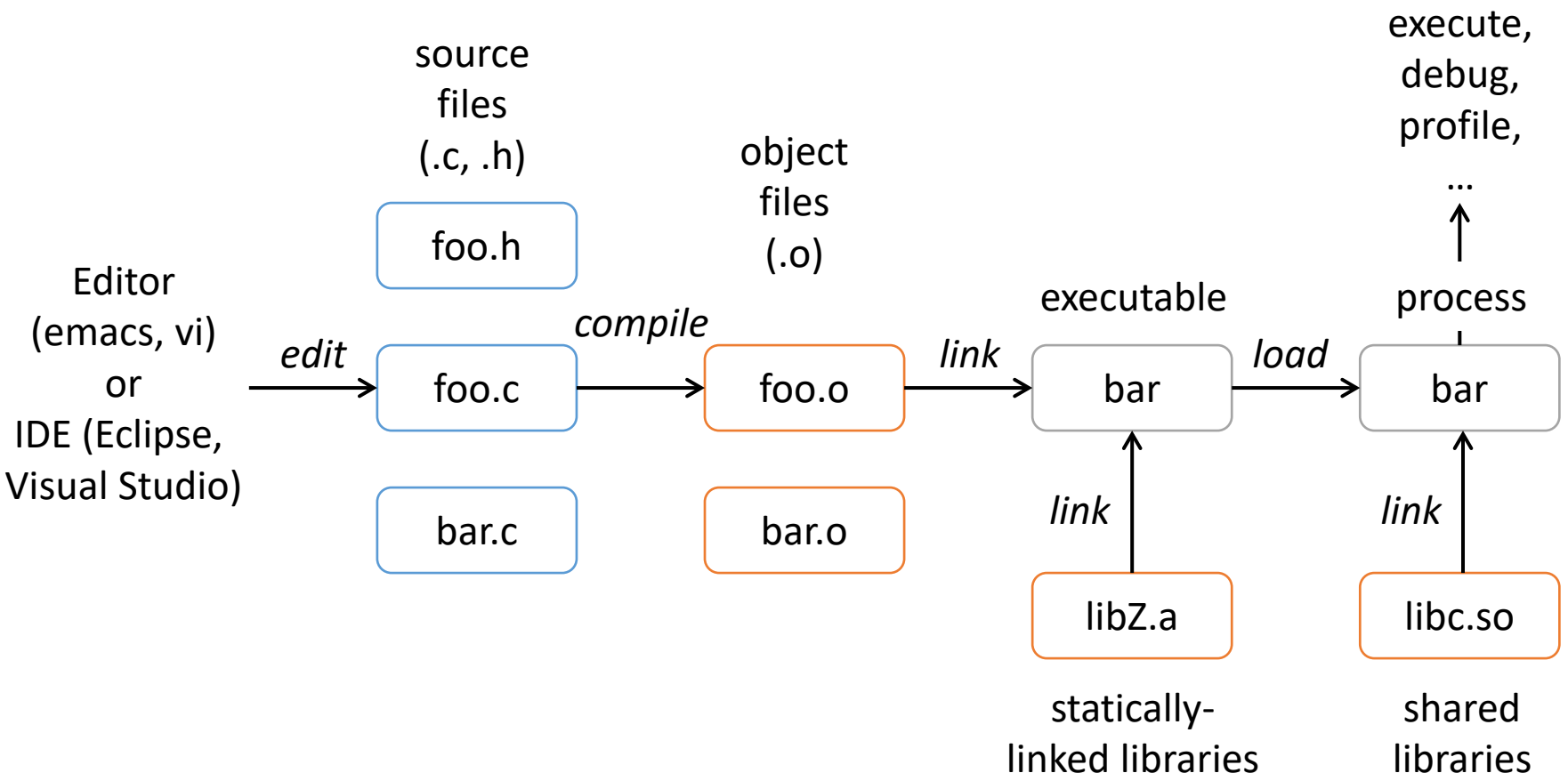
"header file" – bit like an interface file in Java or C#

Every program has to have a "main" function, which takes a list of *command line arguments.*

Generic function for printing formatted strings. The "newline" is not included automatically!

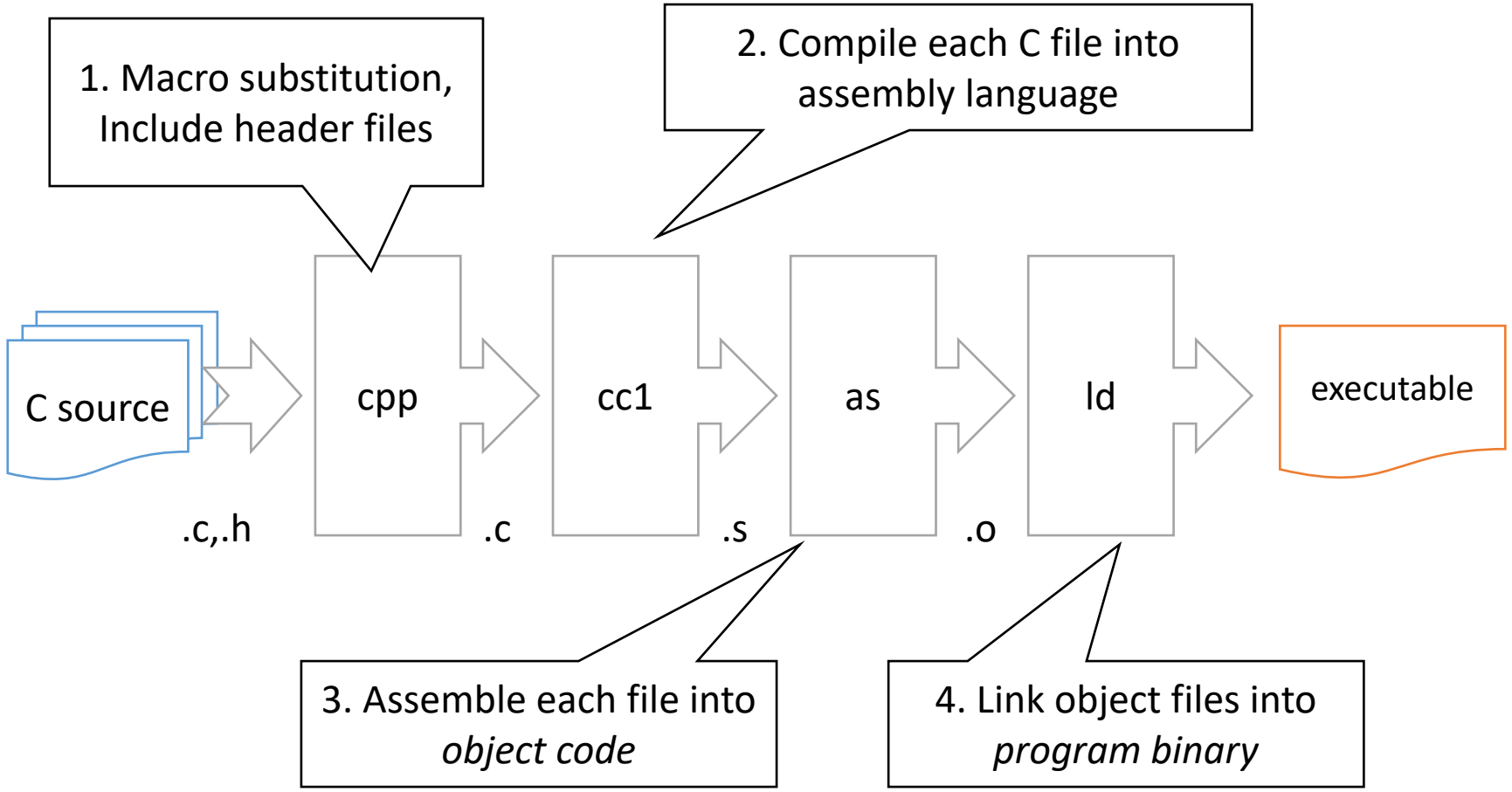
Returning 0 indicates everything is OK – C has no exceptions.

Workflow

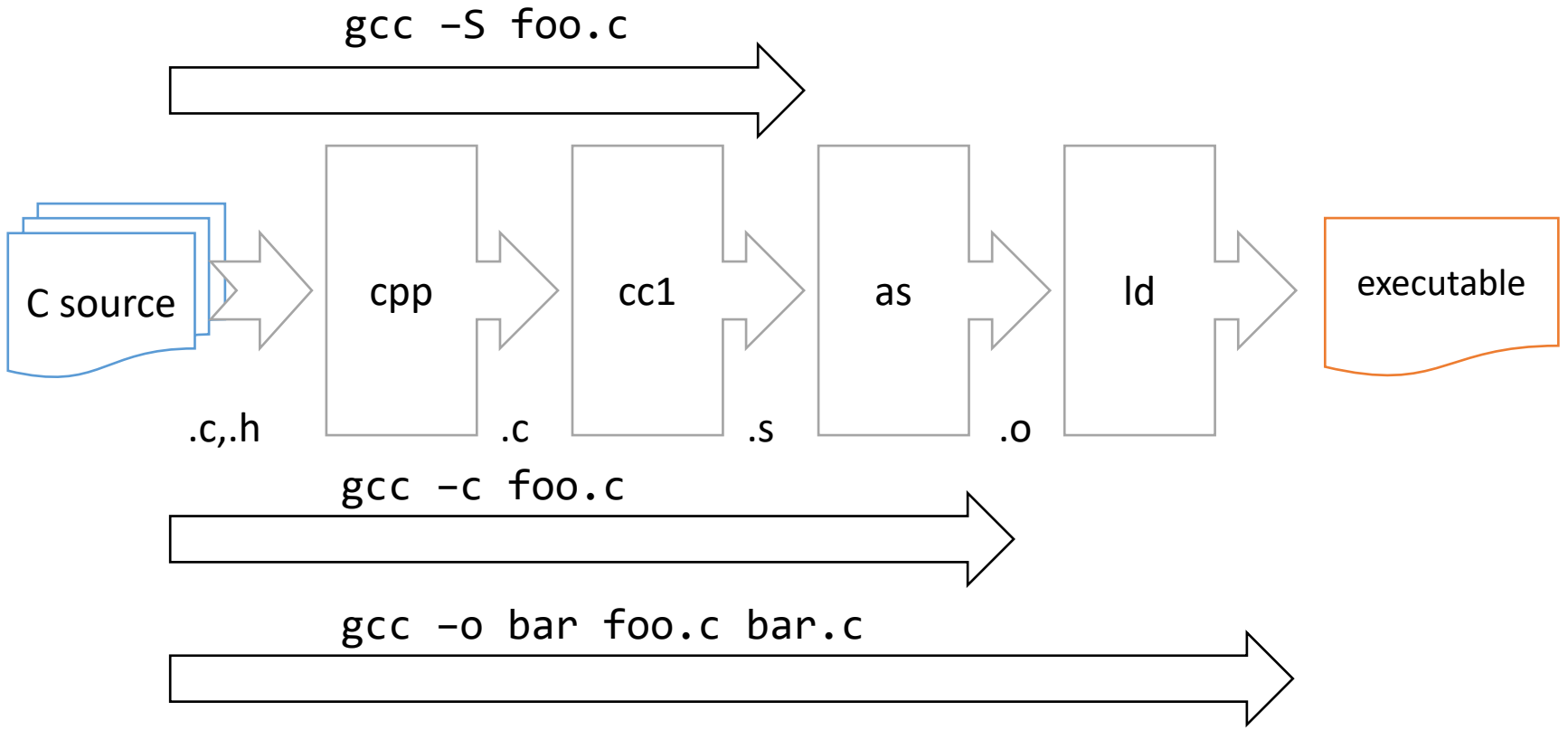


Systems@ETH zürich

GNU `gcc` Toolchain



GNU `gcc` Toolchain



Summary

- C is a *systems programming language!*
  - It is there to program the system.
  - Also useful for high performance

- Understanding C is about understanding how
  - Your program
  - The C compiler
  - The present computer system

… all interact with each other.

2.2: Control flow in C

Systems Programming and Computer Architecture

Control flow statements (like Java or C# or C++)

```
if (Expression) Statement_when_true
        else  Statement_when_false
```

```
switch (Expression) {
        case Constant_1 : Statement; break;
        case Constant_2 : Statement; break;
        …
        case Constant_n: Statement; break;
        default: Statement; break;
}
```

```
return (Expression)
```

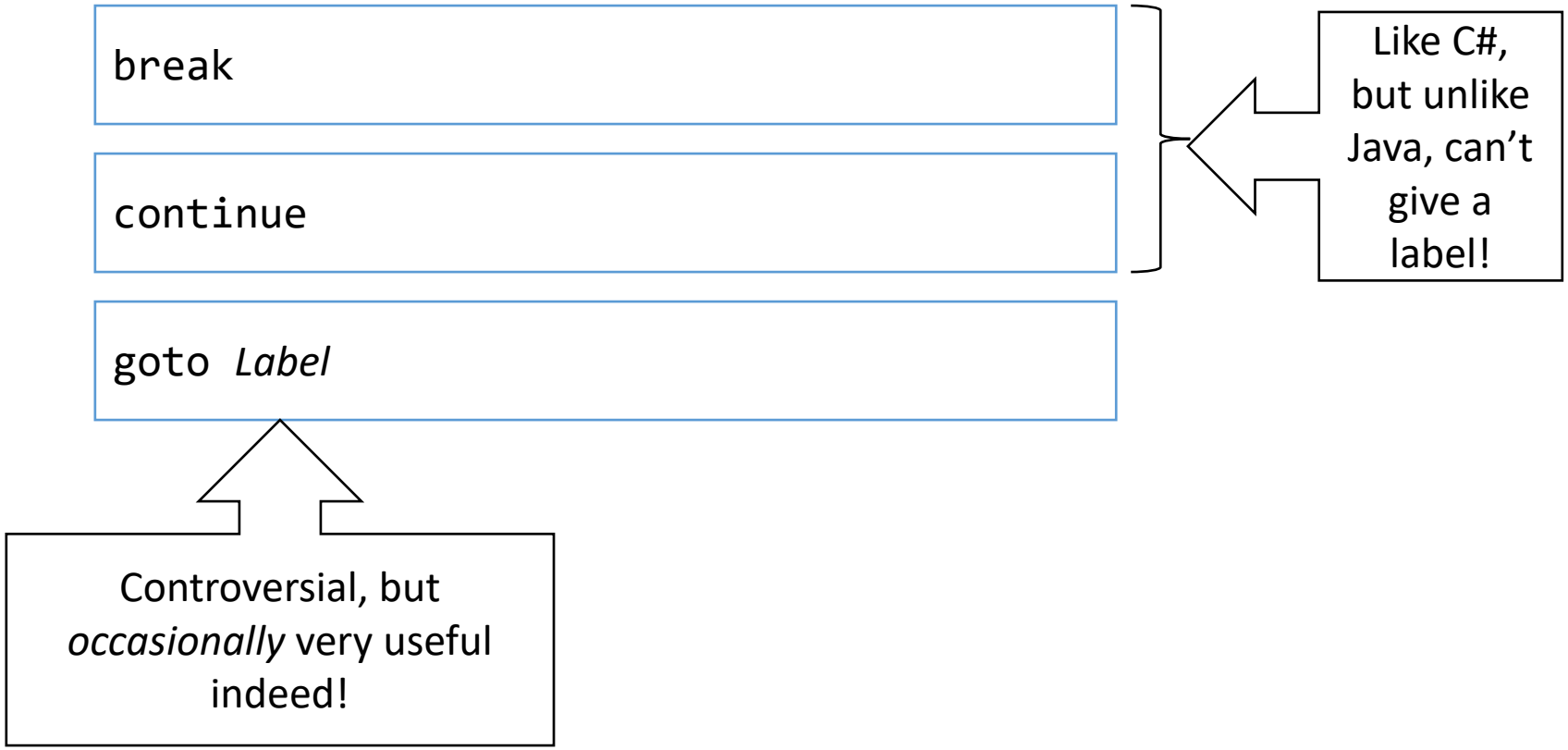
Control flow statements (like Java or C# or C++)

```
for (Initial; Test; Increment) Statement
```

```
while (Expression) Statement
```

```
do Statement while (Expression)
```

Control flow statements (not like Java, same as C#)



Functions: similar to Java

- Name
- Return type
- Argument types
- Body

Source code: chapter 2 → `factorial`

```
#include <stdio.h>

// Compute factorial function
//  fact(n) = n * (n-1) * ... * 2 * 1
int fact(int n)
{
    if (n == 0) {
        return(1);
    } else {
        return(n * fact(n-1));
    }
}

int main(int argc, char *argv[])
{
    int n, m;

    printf("Enter a number: ");
    scanf("%d", &n);
    m = fact(n);
    printf("Factorial of %d is %d.\n", n, m);
    return 0;
}
```

Source code: chapter 2 → `printargs`

`main()`: also a function

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    // Print arguments from command line */
    printf("argc = %d\n\n", argc);
    for (i = 0; i < argc; ++i) {
        printf("argv[%d]: %s\n", i, argv[i]);
    }
    return 0;
}
```

Source code: chapter 2 → `printfdemo`

Basic I/O: `printf()`

- Just another function, but very useful!

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i = 314;
    const char s[] = "Mothy";
    printf("My name is %s and I work in STF H %d\n", s, i);
    return 0;
}
```

`printf` is a *variadic* function!

- First argument is format string
  - see "`man 3 printf`" for all the (many) options
- Remaining arguments are arbitrary
  - but must match the format
- You will see other "printf-like" functions

Summary: control flow in C

- Functions
  - `return (..)`
- Loops
  - `for( ..; ..; ..)`
  - `do .. while (..)`
  - `while (..) ..`
- Conditionals
  - `if (..) then .. else ..`
  - `switch (..) case .. : ..; default ..`
- Jumps
  - `break, continue`
  - `goto ..`
- I/O:
  - `printf()`

2.3: Basic types in C

Systems Programming and Computer Architecture

Declarations

- Are like Java:

```
int my_int;
double some_fp = 0.123;
```

- Inside a block:
  - Scope is just the block
  - `static` → value *persists* between calls
- Outside a block:
  - Scope is the *entire program!*
  - `static` → scope limited to the file (compilation unit)

Source code: chapter 2 → `static`

```
#include <stdio.h>

static int j = 0;

int func(int j)
{
    static int i = 0;

    i = i + 1;
    j = j + 1;
    printf("In func: i=%d, j=%d\n", i, j);
    return j;
}


int main(int argc, char *argv[])
{
    int i = 0;

    printf("In main: i=%d, j=%d\n", i, j);
    func(j);
    printf("In main: i=%d, j=%d\n", i, j);
    func(j);
    printf("In main: i=%d, j=%d\n", i, j);
    return 0;
}
```

Integers and floats

- Types and sizes:

C data type	Typical 32-bit	ia32	Intel x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16

Sizes are implementation defined!

- Integers are signed by default
  - use `signed` or `unsigned` to clarify

C99 extended integer types

```
#include <stdint.h>

int8_t          a;
int16_t         b;
int32_t         c;
int64_t         d;

uint8_t         x;
uint16_t        y;
uint32_t        z;
uint64_t        w;
```

Signed integers, precise size in bits

Unsigned integers,

Integers and floats

- Rules for arithmetic on integers and floats are complicated!
  - Implicit conversions between integer types
  - Implicit conversions between floating point types
  - Explicit conversions between anything (casts)

- Behavior is either:
  - Determined by the hardware (*implementation defined*)
  - Was decided by hardware, a long time ago (standardized)
- We'll cover this more later…

Booleans

- Historically, boolean values are just integers
  - False → zero
  - True → anything non-zero
  - Negation (“!”) turns zero into non-zero, and vice-versa
- C99: new `bool` type supported
  - Completely optional, it’s still an integer
  - `#include <stdbool.h>`

Booleans

- Any statement in C is also an expression, hence:

Source code: chapter 2 → booleans

```
#include <stdio.h>
#include <stdlib.h>

#define ERR_OUT_OF_RANGE (-1)

int test(int i)
{
    if (i >= 0 && i < 10) {
        printf("Success: value is %d\n", i);
        return 0;
    } else {
        return ERR_OUT_OF_RANGE;
    }
}

int main(int argc, char *argv[])
{
    int rc;

    if (argc != 2) {
        fprintf(stderr,"Usage: %s <number>\n", argv[0]);
        return 1;
    }
    if ( (rc = test(atoi(argv[1]))) ) {
        fprintf(stderr,"Error: argument was out of range\n");
        return 1;
    }
    return 0;
}
```

Source code: chapter 2 → booleans

Booleans

- Any statement in C is also an expression, hence:

```
#include <stdio.h>

#define ERR_OUT_OF_RANGE (-1)

int main(int argc, char *argv[])
{
    FILE *f;

    if (argc != 2) {
        fprintf(stderr,"Usage: %s <filename>\n", argv[0]);
        return 1;
    }
    if (!(f = fopen( argv[1], "r" ))) {
        perror("Failed to open file");
        return 1;
    }
    printf("Successfully opened %s\n", argv[0]);
    fclose(f);
    return 0;
}
```

`void`

- There is a type called `void`.
- It has no value.
- Used for:
  - Untyped pointers (to raw memory):
    “`void *`”
  - Declaring functions with no return value (procedures)

Summary: C basic types

- Declarations
- Scopes and `static`
- Integers and floats, extended types
- Booleans
- `void`

2.4: Operators

Systems Programming and Computer Architecture

Operators: similar to Java

Decreasing precedence

Operator	Associativity
() [] -> .	Left-to-right
! ~ ++ -- + - * & (type) sizeof	Right-to-left
* / %	Left-to-right
+ -	Left-to-right
<< >>	Left-to-right
< <= > >=	Left-to-right
== !=	Left-to-right
&	Left-to-right
^	Left-to-right
|	Left-to-right
&&	Left-to-right
||	Left-to-right
?:	Right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Right-to-left
,	Left-to-right

- `()` here is a function call
- `->` means *struct pointer indirection*

- Unary `+, -, *`
- `*` here is *pointer indirection*

- Ternary if-else operator

- Assignment operators

Assignment operators

- In many imperative languages

```
x = foo();
```

  - is an assignment **statement**.

- In C, it is an **expression!**
  - Value is the value being assigned

- Also:

```
x += y   ↔   x = x + y
```

- and so with -=, *=, <<=, etc.

What is associativity again?

- Left-to-right associativity:
  - A + B + C → (A + B) + C
  - A + B + C + D → ((A + B) + C) + D
- Right-to-left:
  - A += B += C → A += (B += C)
  - Makes sense here, but elsewhere it's rarely what you want…

Post-increment and pre-increment

- `i++`
  - Value: current value of `i`
  - Effect: `i` ← `i+1`
- `++i`
  - Effect: `i` ← `i+1`
  - Value: new value of `i`
- Conversely `i--` and `--i`
- Works for any scalar type
  - Importantly: works for pointers!

Historical:
Digital PDP computers had pre- and post-increment and -decrement addressing modes

Casting

- Most C types can be *cast* to another:

```
unsigned int ui = 0xDEADBEEF;
signed int i    = (signed int)ui;
```

⇒ `i` has value -559038737.

Name of type in parentheses functions like an operator.

- Bit-representation does (usually…) not change
- Frequently used with pointer types…

Summary: C operators

- Operators and precedence
- Assignment operators
- Post/pre inc/decrement
- Casting

2.5: Arrays in C

Systems Programming and Computer Architecture

Source code: chapter 2 → array

Arrays

- Finite vector of variables, all the same type
- For an N-element array a:
  - First element is `a[0]`
  - last element is `a[N-1]`
- C compiler does not check the array bounds!
  - Very typical bug!
  - Always check array bounds!

```c
#include <stdio.h>

float data[5];  // data to average and total
float total;    // total of the data items
float average;  // average of the items

int main() {
    data[0] = 34.0;
    data[1] = 27.0;
    data[2] = 45.0;
    data[3] = 82.0;
    data[4] = 22.0;

    total = data[0] + data[1] + data[2] + data[3] + data[4];
    average = total / 5.0;
    printf("Total %f Average %f\n", total, average);
    return (0);
}
```

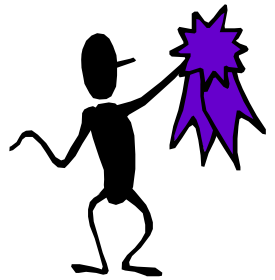
Multi-dimensional arrays

```
int a[3][3];
```

0	0	0	0	0	0	0	0	0

a[0][0] … a[1][0] … a[1][2] … a[2][2]

```
int a = 1;
for (i=0; i < 3; i++)
        for (j=0; j < 3; j++)
                matrix[i][j] = a++;
```



1	2	3	4	5	6	7	8	9

```
int a = 1;
for (i=0; i < 3; i++)
        for (j=0; j < 3; j++)
                matrix[j][i] = a++;
```

1	4	7	2	5	8	3	6	9

Source code: chapter 2 → array

Array initializers

- Arrays can be initialized when they are defined:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i, j;
    int a[3] = {3, 7, 9};
    int b[3][3] = {
         { 1, 2, 3 },
         { 4, 5, 6 },
         { 7, 8, 9 },
    };

    for(i = 0; i < 3; i++) {
         printf("a[%d] = %d\n", i, a[i] );
         for(j = 0; j < 3; j++) {
             printf("  b[%d][%d] = %d\n", i, j, b[i][j] );
         }
    }

    return 0;
}
```

Source code: chapter 2 → array

Array initializers

- Arrays can be initialized when they are defined:

```
#include <stdio.h>

#define ARRAY_SIZE 10

int main(int argc, char *argv[])
{
    int i;

    float lu[ARRAY_SIZE];       // Uninitialized
    float li[ARRAY_SIZE] = {};  // Initialized to 0.0

    for(i = 0; i < ARRAY_SIZE; i++) {
        printf("li[%d] = %f, lu[%d] = %f\n", i, li[i], i, lu[i]);
    }
    return 0;
}
```

Source code: chapter 2 → `strings`

Strings

- C has no real string type!
  - Instead…
- Array of `char`'s terminated with null byte
  - `0` or `'\0'`

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    // These strings are identical
    char s1[6] = "hello";
    char s2[6] = { 'h', 'e', 'l', 'l', 'o', 0 };

    printf("s1 = '%s'\n", s1);
    printf("s2 = '%s'\n", s2);

    for( i = 0; i < 6; i++ ) {
        printf("s1[%d] = %d, s2[%d] = %d\n", i, s1[i], i, s2[i]);
    }
    return 0;
}
```

Source code: chapter 2 → `string`

String library functions

Generally named '`str`*`xxx`*`()`'

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char name1[12], name2[12];
    char mixed[25], title[20];

    strcpy(name1, "Rosalinda");
    strcpy(name2, "Zeke");
    strcpy(title, "This is the title.");

    printf("     %s\n\n", title);
    printf("Name 1 is %s\n", name1);
    printf("Name 2 is %s\n", name2);

…
```

```
…

    /* returns 1 if name1 > name2 */
    if (strcmp(name1, name2) > 0) {
        strcpy(mixed, name1);
    } else {
        strcpy(mixed, name2);
    }
    printf("The biggest name alphabetically is

    strcpy(mixed, name1);
    strcat(mixed, "  ");
    strcat(mixed, name2);
    printf("Both names are %s\n", mixed);
    return 0;
}
```

```
     This is the title.

Name1 is Rosalinda
Name2 is Zeke
The biggest name
alphabetically is Zeke
Both names are Rosalinda
Zeke
```

Summary: C arrays

- Arrays of basic types
- Multidimensional arrays
- Initializers
- Strings
  - Arrays of ASCII characters
  - Null-terminated
- String library

We'll see a lot more C as the course progresses…