SWEN-250 Personal SE

Introduction to C

A Bit of History

- Developed in the early to mid 70s
  - Dennis Ritchie as a systems programming language.
  - Adopted by Ken Thompson to write Unix on a the PDP-11.
- At the time:
  - Many programs written in assembly language.
  - Most systems programs (compilers, etc.) in assembly language.
  - Essentially ALL operating systems in assembly language.
- Proof of Concept
  - Even small computers could have an OS in a HLL.
  - Small: 64K bytes, 1µs clock, 2 MByte disk.
  - We ran *5 simultaneous users* on this base!

But Efficiency Wasn't Cheap in the 70s

- Code written in assembly
- High level languages in their infancy
- Desire to write programs with fewer lines of code, but retain control
- C as a consequence:
  - Has types (but they can be easily ignored).
  - Has no notion of objects (just arrays and structs)
    - OO was a mostly a research topic
  - Permits pointers to arbitrary locations in memory (
  - Has no garbage collection – it's the programmer's job to manage memory.
- C was a major advancement from FORTRAN, MACRO ASSEMBLER, BUT:
  - Very powerful and doesn't get in your way.
  - Very dangerous and you can cut off your fingers.

Most languages have borrowed from C

- { and } for grouping.
- Prefix type declaration (e.g., int i vs. i : int).
- Control structures (mostly)
  - if, switch
  - while, for
- Arithmetic (numeric) operations:
  - ++ and -- (prefix and suffix)
  - *op*= (e.g. += *=, etc.)
  - + - * / %
- Relational & boolean operators:
  - < > <= >= != ==
  - ! || &&

- C++
- Java
- C#
- Javascript
- PHP
- …

Things Uniquely C vs. Interpreted languages

- Today
  - No classes – just functions & data.
  - Characters are just small integers.
  - No booleans. *
  - Limited visibility control via #include and separate compilation.
  - Simple manifest constants via #define
- Later
  - Array size fixed at compile time.
  - Strings are just constant arrays.
  - Simple data aggregation via structures (**struct**)
  - And, last but not least – POINTERS!!!

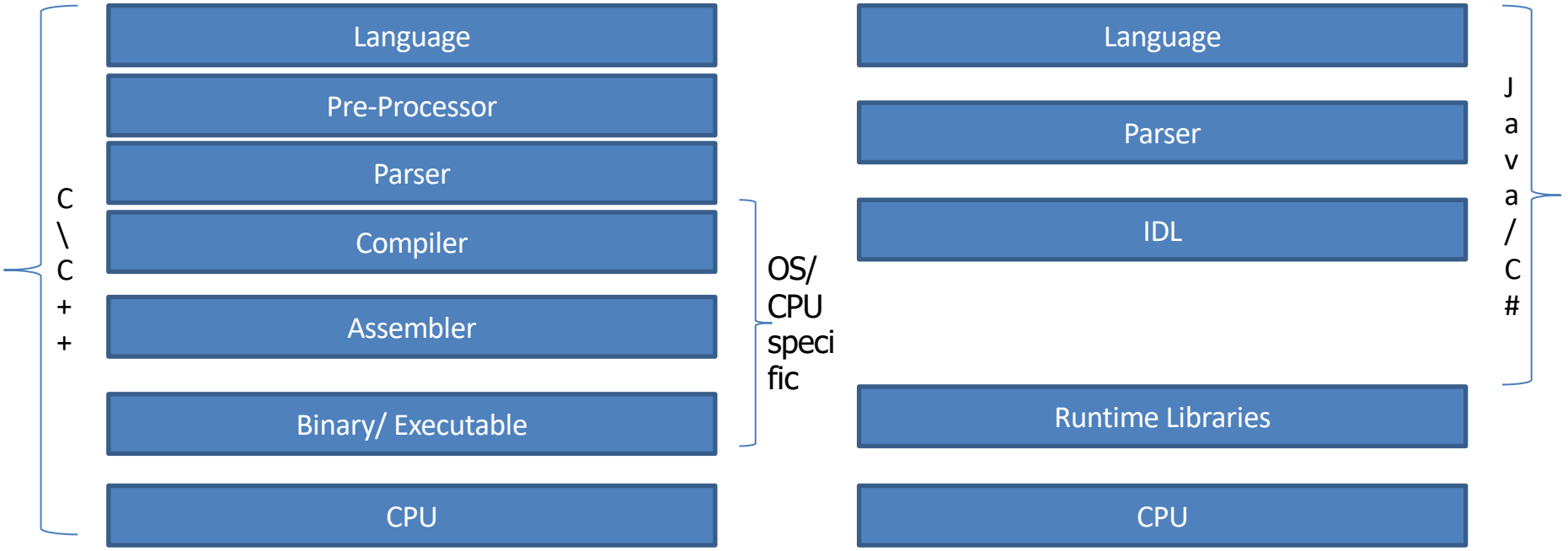
*In the C99 version, there is '_Bool'. However C99 is not universally adopted.

Compiled vs. Interpreted

- Short version
  - Compiled languages are converted to CPU specific binary code and then run (C/ C++/ FORTRAN/ Eiffell, PL-I …)
  - Interpreted languages are converted to intermediate ‘bytecode’ and run within a runtime library which is specific to each CPU/ OS (Java, C#, Ruby, …)

Software Engineering
Rochester Institute of Technology

Compiled vs. interpreted languages



For 'C', you will need to execute a command like
gcc –o <outputfile> <inputfile.c>

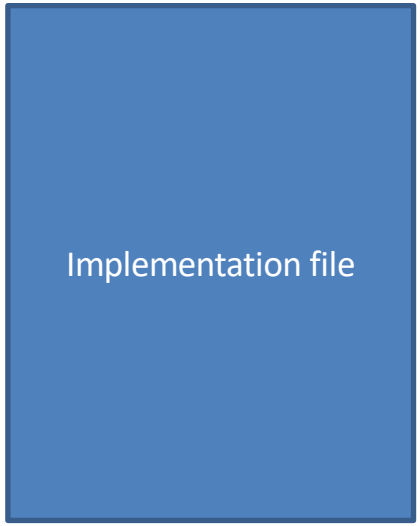
Software Engineering
Rochester Institute of Technology

Basics: 2 file approach

.h file (header)

'include' this file to reference:
- Variables
- Functions
- Classes

.c/ .cpp file on Windows .c/ .cc file on *nix



Your implementation code goes here

In very, very trivial programs (i.e. just a few line of code in 'main', you may get away with not adding a `.h file)

stdin and stdout

- You will typically work from the command line (console)
- stdin is ‘standard in(put)’
  - This is where C will assume any incoming data is ‘input’ from. Usually the command line, but often used via redirection from a file
- stdout is ‘standard out(put)’
  - Normally output (from printf or puts) goes to the console, but can also be redirected

Functions & Data

- C functions – like methods free from their class.
- The most important function: main
- Example: Hello, world

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

int main( ) {
    puts( "Hello, world!" ) ;
    return 0 ;
}
```

Functions & Data

- C functions – like methods free from their class.
- The most important function: main
- Example: Hello, world

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

int main( ) {
    puts( "Hello, world!" ) ;
    return 0 ;
}
```

Includes interface information to other modules

Similar to import in Java

But done textually!!

Functions & Data

- C functions – like methods free from their class.
- The most important function: main
- Example: Hello, world

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

int main( ) {
    puts( "Hello, world!" ) ;
    return 0 ;
}
```

stdlib
atoi, atol, atof
memory allocation
abort, exit, system, atexit
qsort, bsearch [advanced]

Functions & Data

- C functions – like methods free from their class.
- The most important function: main
- Example: Hello, world

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

int main( ) {
    puts( "Hello, world!" ) ;
    return 0 ;
}
```

stdio
getchar, fgetc, putchar, fputc
printf, fprintf, sprintf
gets, puts, fgets, fputs
scanf, fscanf, sscanf

Functions & Data

- C functions – like methods free from their class.
- The most important function: main
- Example: Hello, world

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

int main( ) {
    puts( "Hello, world!" ) ;
    return 0 ;
}
```

Every C program has a **main** function – the first function called.

main returns exit status.
0 = ok
anything else = abnormal.

Functions & Data

- C functions – like methods free from their class.
- The most important function: main
- Example: Hello, world

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

int main( ) {
    puts( "Hello, world!" ) ;
    return 0 ;
}
```

puts, from **stdio**, prints a string and appends a newline ('\n').

Strings are simpler in C than Java.

C strings are just arrays of characters.

Comments

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

/*This is a comment*/

int main( ) {
    puts( "Hello, world!" ) ;
    return 0 ;
}
```

Printing to the console

- The ‘C’ function printf can also be used to print strings or other data

```
printf("Hello printf world\n");
printf("%s\n","Hello %s");
int i = 5;
printf("Value of i is %d\n",i);
```

Note the special characters for \n and %s, %d

Note that variables are declared with the data type! (int i;)

Flow control and iteration

Flow control in 'C' uses normal 'if then else' syntax

```
if (value > 5)
{
  printf("It's big\n");
}
else
{
  printf("It's small\n");
}
```

Simple for loops look like this

```
for (int i = 0; i < 5; i++)
{
  printf("I = %d\n", i);
}
OR
for (int i = 0; i < 22; i+=2)
{
  printf("I = %d\n", i);
}
```

Watch for compiler differences. You may need to declare your loop variable OUTSIDE the for loop!

Characters are ASCII Bytes

- Consider the following C constants"

  `'a'  97(decimal)  0141(octal)  0x61(hex)`

- In C they are all the *same value* – a small positive **integer**.
- That is, character constants are just small integers.
  - Use the notation that expresses what you are doing:
  - If working with numbers, use 97 (or 0141 / 0x61 if bit twiddling).
  - If working with letters, use 'a'.
  - Question: what is 'a' + 3?
  - Question: if ch holds a lower case letter, what is ch - 'a'?
- Escape sequences with backslash:
  - `'\n'` == newline, `'\t'` == tab, `'\r'` == carriage return
  - `'\`*ddd*`'` == character with octal code *ddd* (the *d*'s are digits 0-7).
  - `'\0'` == NUL character (end of string in C).

Integer Types in C

• char	one byte = 8 bits - possibly signed
• unsigned char	one byte unsigned
• short	two bytes = 16 bits signed
• unsigned short	two bytes unsigned
• int	"natural" sized integer, signed
• unsigned int = unsigned	"natural" sized integer, unsigned
• long	four bytes = 32 bits, signed
• unsigned long	four bytes, unsigned
• long long	eight bytes = 64 bits, signed
• unsigned long long	eight bytes, unsigned

Another Example – Count Punctuation

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

int main( ) {
    int tot_punct = 0 ; // declare & init. a local variable
    int nchar ;         // next character read

    while( (nchar = getchar()) != EOF ) {
        if( ispunct(nchar) ) {
                tot_punct++ ;
        }
    }

    printf( "%d punctuation characters\n", tot_punct ) ;
    return 0 ;
}
```

Another Example – Count Punctuation

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

int main( ) {
    int tot_punct = 0 ; // declare & init. a local variable
    int nchar ;         // next character read

    while( (nchar = getchar()) != EOF ) {
        if( ispunct(nchar) ) {
            ++tot_punct ;
        }
    }

    printf( "%d punctuation characters\n", tot_punct ) ;
    return 0 ;
}
```

ctype
isalnum, isalpha, isdigit, iscntrl
islower, isupper, ispunct, isspace
isxdigit, isprint
toupper, tolower

Another Example – Count Punctuation

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

int main( ) {
    int tot_punct = 0 ; // declare & init. a local variable
    int nchar ;         // next character read

    while( (nchar = getchar()) != EOF ) {
        if( ispunct(nchar) ) {
            ++tot_punct ;
        }
    }

    printf( "%d punctuation characters\n", tot_punct ) ;
    return 0 ;
}
```

Next character from standard in.
Why **int** and not **char**?
Because EOF is negative!

Another Example – Count Punctuation

Common C idiom:
Get & assign value
Compare to control flow
= vs. == can kill you here.

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

int main( ) {
    int tot_punct = 0 ; // declare & init. a local variable
    int nchar ;         // next character read

    while( (nchar = getchar()) != EOF ) {
        if( ispunct(nchar) ) {
            ++tot_punct ;
        }
    }

    printf( "%d punctuation characters\n", tot_punct ) ;
    return 0 ;
}
```

Another Example – Count Punctuation

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

int main( ) {
    int tot_punct = 0 ; // declare & init. a local variable
    int nchar ;         // next character read

    while( (nchar = getchar()) != EOF ) {
        if( ispunct(nchar) ) {
            ++tot_punct ;
        }
    }

    printf( "%d punctuation characters\n", tot_punct ) ;
    return 0 ;
}
```

EOF defined in **stdio.h** as (-1)
Not a legal character.
Signals end-of-file on read.

Wait, what file??

Another Example – Count Punctuation

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

int main( ) {
    int tot_punct = 0 ; // declare & init. a local variable
    int nchar ;         // next character read

    while( (nchar = getchar()) != EOF ) {
        if( ispunct(nchar) ) {
            ++tot_punct ;
        }
    }

    printf( "%d punctuation characters\n", tot_punct ) ;
    return 0 ;
}
```

Helper function from **ctype**
True iff nchar is punctuation.

Another Example – Count Punctuation

Formatted output to standard out.
printf = **print f**ormatted
1st argument is format string
Remaining arguments are printed according to the format.

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

int main( ) {
    int tot_punct = 0 ; // declare & init. a local variable
    int nchar ;         // next character read

    while( (nchar = getchar()) != EOF ) {
        if( ispunct(nchar) ) {
            ++tot_punct ;
        }
    }

    printf( "%d punctuation characters\n", tot_punct ) ;
    return 0 ;
}
```

Short Digression on Printf

- Format string printed as is except when encounters '%'
  - %d print integer as decimal
  - %f print floating point (fixed point notation)
  - %e print floating point (exponential notation)
  - %s print a string
  - %c print integer as a character
  - %o / %x print integer as octal / hexadecimal
- Format modifiers - examples
  - %***n***.***m***f at least ***n*** character field with ***m*** fractional digits
  - %***n***d at least ***n*** character field for a decimal value.
- Example:

```
printf("%d loans at %5.2f%% interest\n",nloans, pct) ;
```

- See the stdio.h documentation for more on format control.

Boolean = Integer

- There is no boolean type in C.*
- 0 is **false**, *everything* else is **true**.
  - False: 0 0.0 '\0' NULL (0 pointer).
  - True: 1 'a' 3.14159
- The result of a comparison operator is 0 or 1.
- Many programmers define symbolic constants:

```
#define TRUE  (1)
#define FALSE (0)
```

- Pet Peeve:

VERY BAD

```
return value < limit;
```

SLOPPY

```
if ( value < limit )
   return TRUE;
else
    return FALSE;
```

GOOD PRACTICE

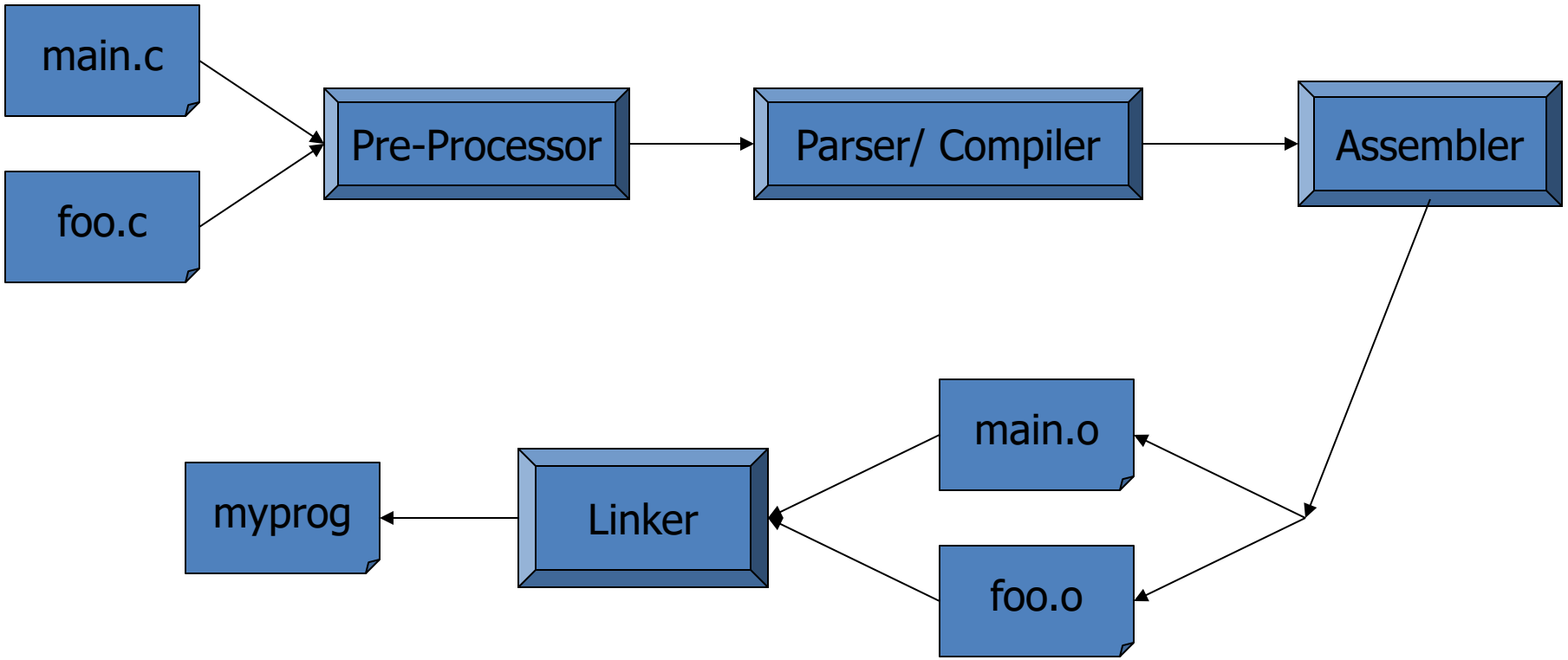
```
int result = FALSE;
if ( value < limit )
   result = TRUE ;
return result;
```

*In the C99 version, there is '_Bool'. However C99 is not universally adopted.

Compilation

- Our systems use the GNU C compiler (gcc)
- The compilation process with two files (main.c, foo.c)

```
gcc –o myprog main.c foo.c
```



Compilation

- Problems can occur all along the line:
  - Unterminated comments can throw off the lexer.
  - Syntax errors are detected by the parser.
  - The code generator / optimizer can generate bad code (highly unlikely).
  - The linker may not be able to resolve all the external references.
- Notes on linking:
  - Every object file has a table of contents.
  - Some of the names are defined in the file (e.g., main).
  - Some are needed from another file (e.g., printf).
  - The linker tries to resolve these BUT:
    - It may not be able to find a symbol it needs (missing file?)
    - It may find two definitions of a symbol (name conflict).