Unix
Bash
C
GNU
Systems

Software Systems

Lectures Week 5

Introduction to C

Prof. Joseph Vybihal

Computer Science

McGill University

Week 5 Lecture 1

Class Test

Week 5 Lecture 2

Introduction to C

Readings: chapter 3, https://www.tutorialspoint.com/cprogramming/ or http://www.w3schools.in/c-tutorial/intro/

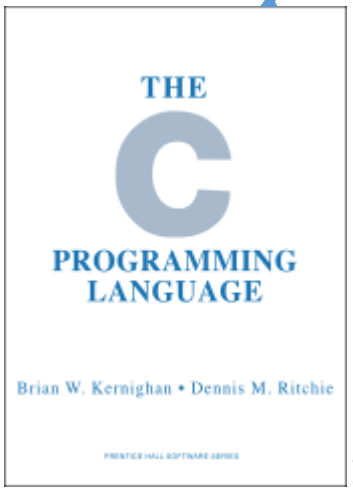
History of C

Denis Ritchie
1941 – 2011

Algol	• International Group
BCPL	• Martin Richards
B	• Ken Thomson
Traditional C	• Dennis Ritchie
K&R C	• kernighan & Ritchie
ANSI C	• ANSI Commitee
ANSI/ISO C	• ISO Commitee
C99	• Standerd Commitee

The B language:
- Interpreted C
- Very slow

1972 AT&T Bell Labs



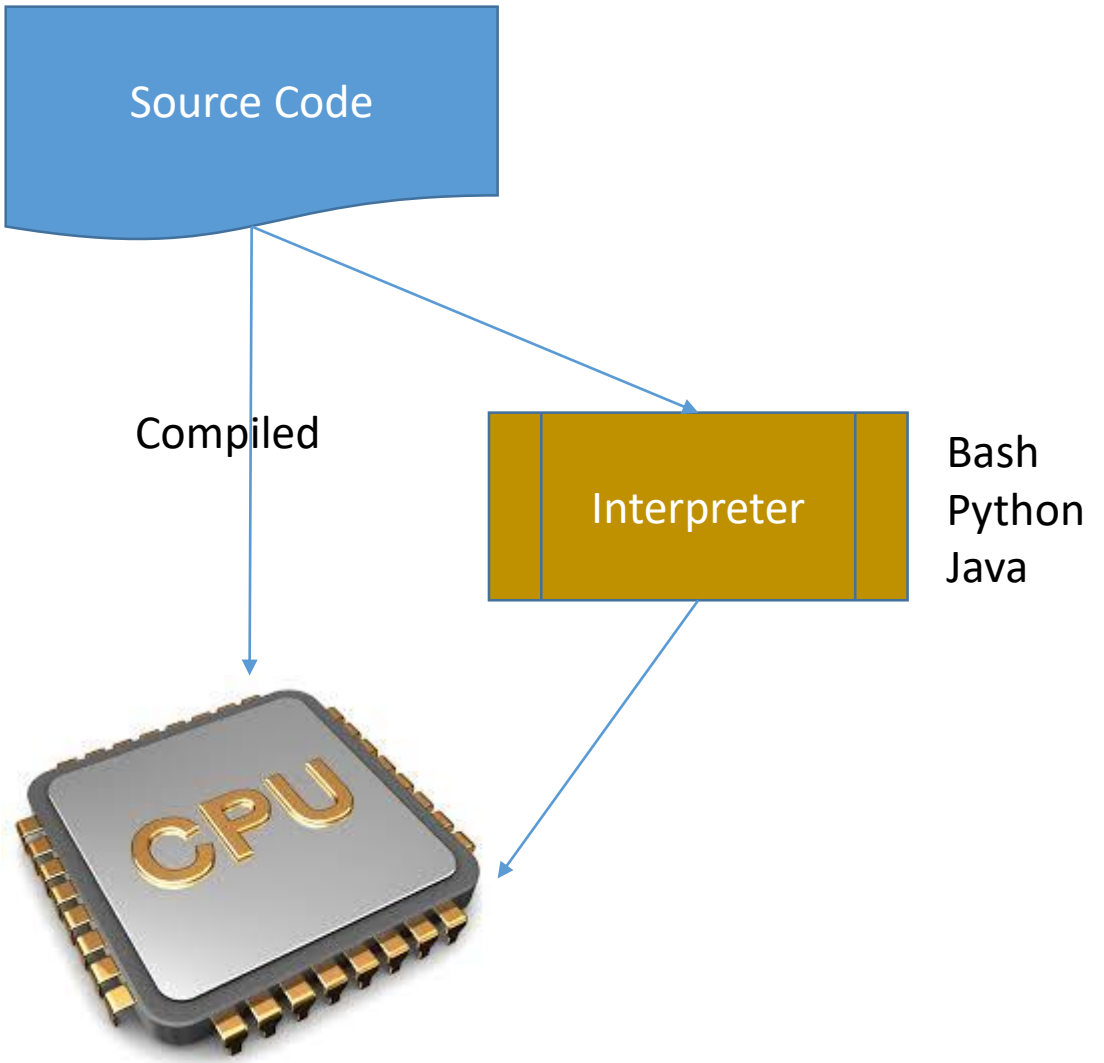
1978

The C language:
- Compiled C
- Created to build the Unix OS

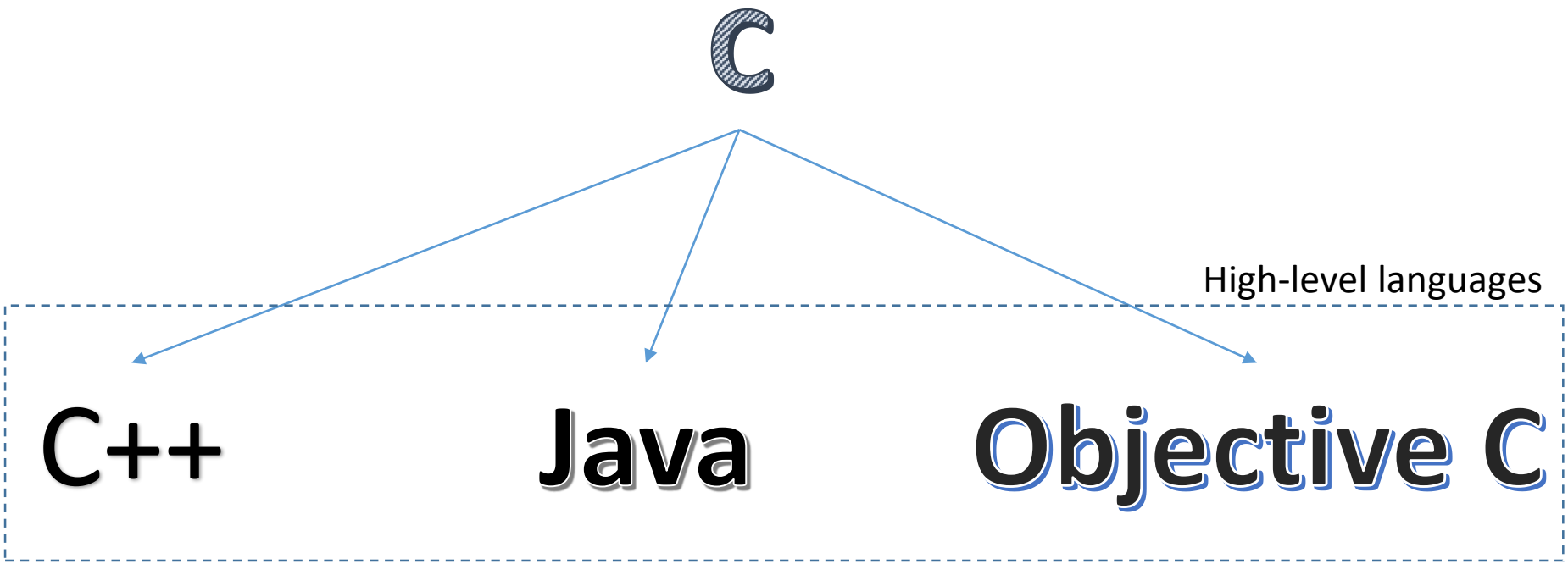
Compilers vs Interpreters

Notice how a compiled program can speak directly with the CPU.

This gives it additional speed and low-level connectivity.



The children of C



Compiled.
Object oriented.
Preserves low-level features.

Highly optimized interpreter.
Object oriented.
No low-level features.
Cross-platform.

1980's improvement of C used by Apple to build their OS X operating system.
Like Small-talk and C++.

Why C?

Because we need an "easy" language that can talk to the hardware and the human.

- Operating systems
- Hardware drivers: printers, mice, etc.
- Specialty machine connectivity: lab machines, robots, VR, etc.

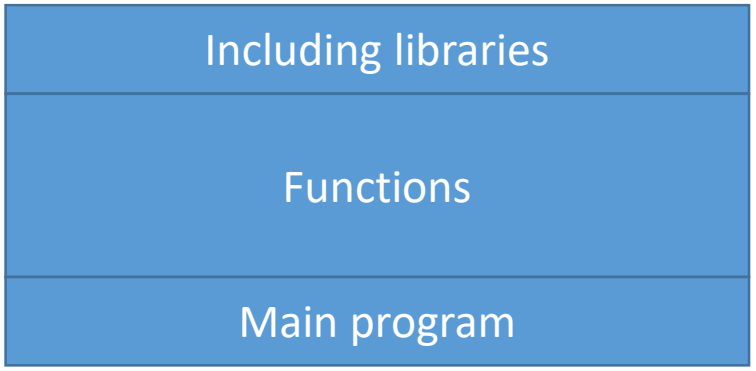
Assembler (COMP 273) is much better but also much harder to write programs.

Basic Structure of a C Program

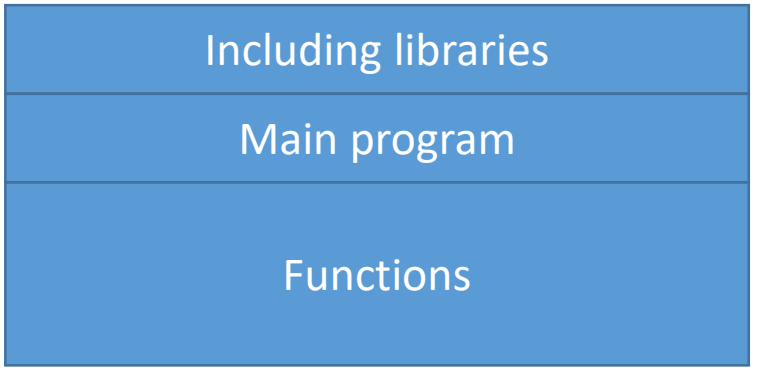
```
#include <stdio.h>
int main(void)
{
  puts("Hello World\n");

   return 0;
}
```

Library

Main program



Recommended layout



Archaic layout

How to compile and run a C program

Bash-prompt $ vi helloworld.c

Bash-prompt $ gcc helloworld.c

Bash-prompt $ ./a.out

- We use VI to create out programs
- The GCC compiler is a powerful tool to convert text files into binary machine-code files
- The a.out file is the default binary machine code file name
  - Also known as the Executable file
  - Executable files speak directly with the CPU
- Notice that we execute a.out the same way we executed Bash files, using the ./

Demo

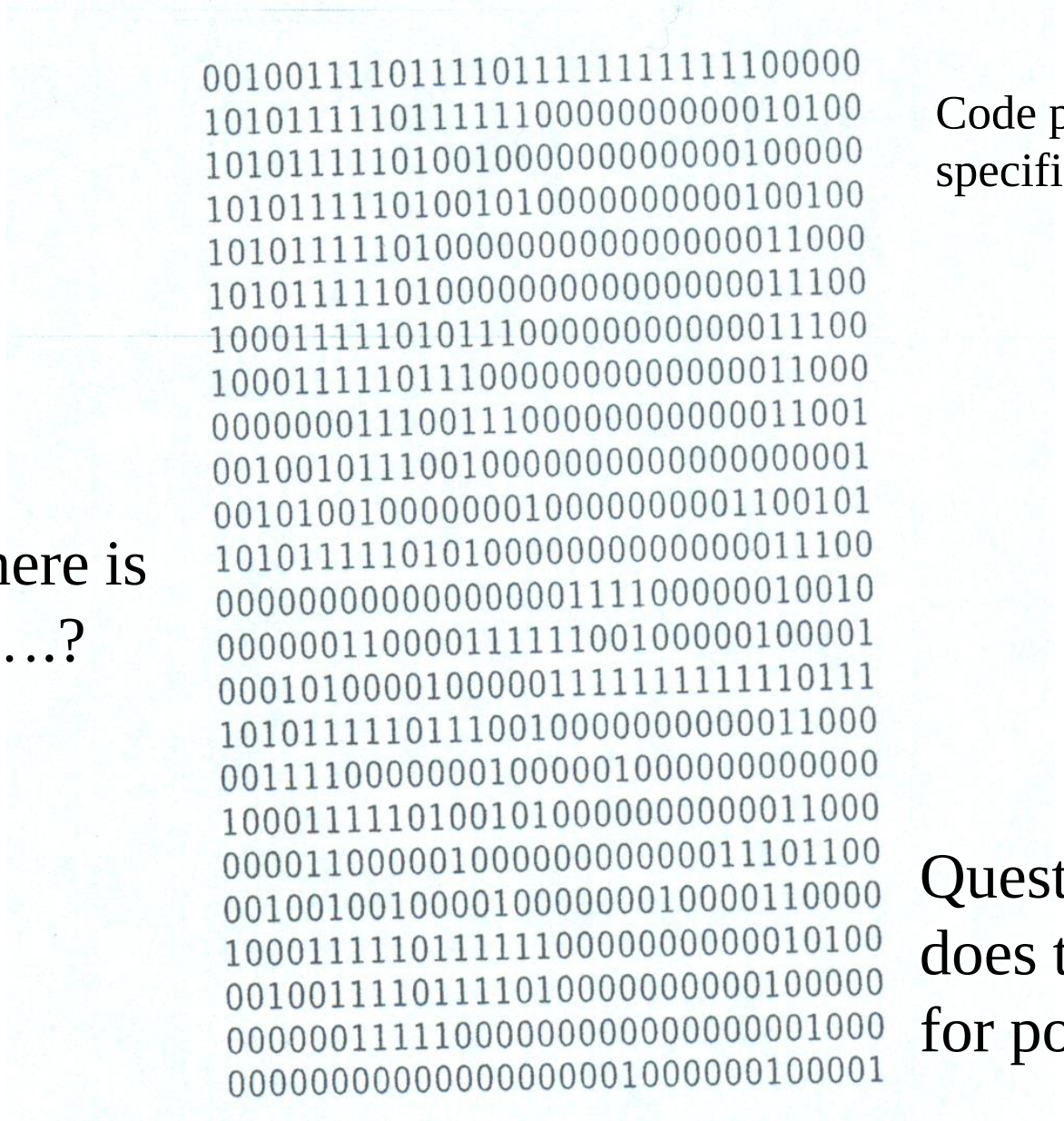
Intel Assembly

```
main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
andl $-16, %esp
movl $0, %eax
subl %eax, %esp
subl $12, %esp
pushl $.LC0
call puts
addl $16, %esp
movl $0, %eax
leave
ret
```

```
#include <stdio.h>
int main(void)
{
  puts("Hello World\n");
  return 0;
}
```

Library call

Machine Code



Hmm, where is my error….?

Code pattern is specific to CPU

Question: what does this mean for portability?

Basic Structure of a C Program

```
#include <stdio.h>

int main(void)
{
   char c;

   puts("Gender: ");
   c = getc(stdin);

   if (c == 'F' || c == 'f')
    puts("Welcome\n");
   else
    puts("Sorry, try again.\n");

   return 0;
}
```

STDIO.H is the standard input/output library.
- Function puts() writes strings.
- Function getc() reads a character.

Declaring and using variables.

Returning error codes like Bash.

puts

Library: stdio.h

Syntax: int puts(constant_string);

Returns: Error code

- >= 0 if no error

Purpose: To print a string to standard out

Usage:

```
puts("Hello World");     // without new line
puts("Hello World\n");   // with new line
```

Escape Characters

\n - New line

\r - Carriage return

\t - Tab

\\ - Backslash

\a - Bell

\b - Backspace (without delete)

Others…

getc

Library: stdio.h

Syntax: int getc(stdin);

Returns: ASCII code

Purpose: To read a character from standard in

Usage:

```
c = getc(stdin);
```

Notice that this functions does not actually return the character but the ASCII code for that character as an integer number.

This is a low-level feature.

gcc

GNU C Compiler

- gcc SWITCHES FILES

Switches

- Without a switch the default activity is to merge all the FILES into a single a.out executable file.
- -o Replace the default a.out file name with your own
  - gcc –o hello helloworld.c

```
Bash-prompt $ gcc –o hello helloworld.c
Bash-prompt $ ./hello
```

GCC and Errors

Errors are displayed to the screen and can be lost as the screen scrolls.

Solution: gcc helloworld.c > textfilename

All output from gcc will be stored in the textfilename file. You can then use vi, more, or cat to view the contents.

Demo

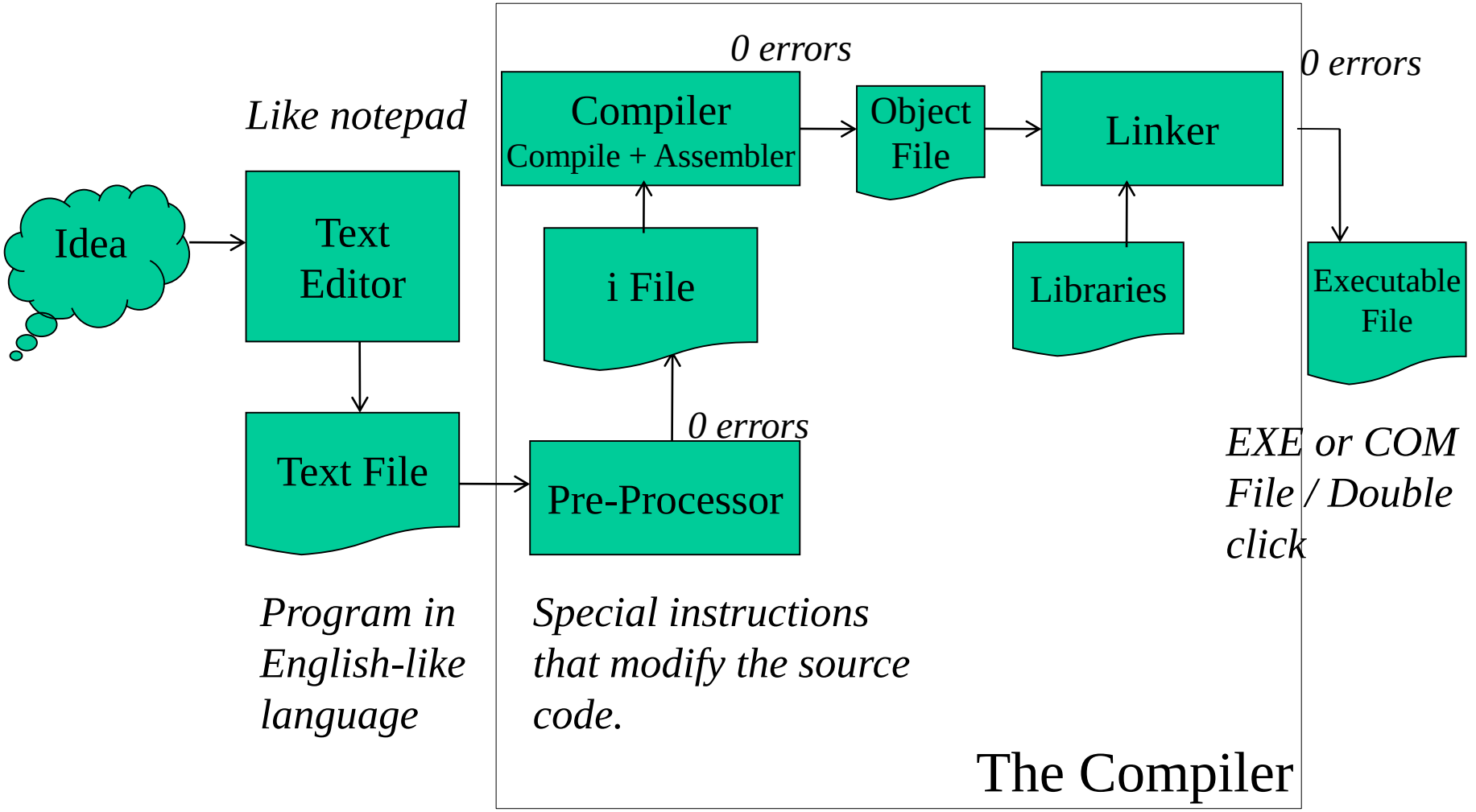
How compiler errors work

A compiler attempts to convert your source code to machine code.

When it finds an error it marks it as an error BUT then makes an assumption and continues compiling.

All other errors are based on the assumption. Trust only the first couple of errors.

The C Compilation Process

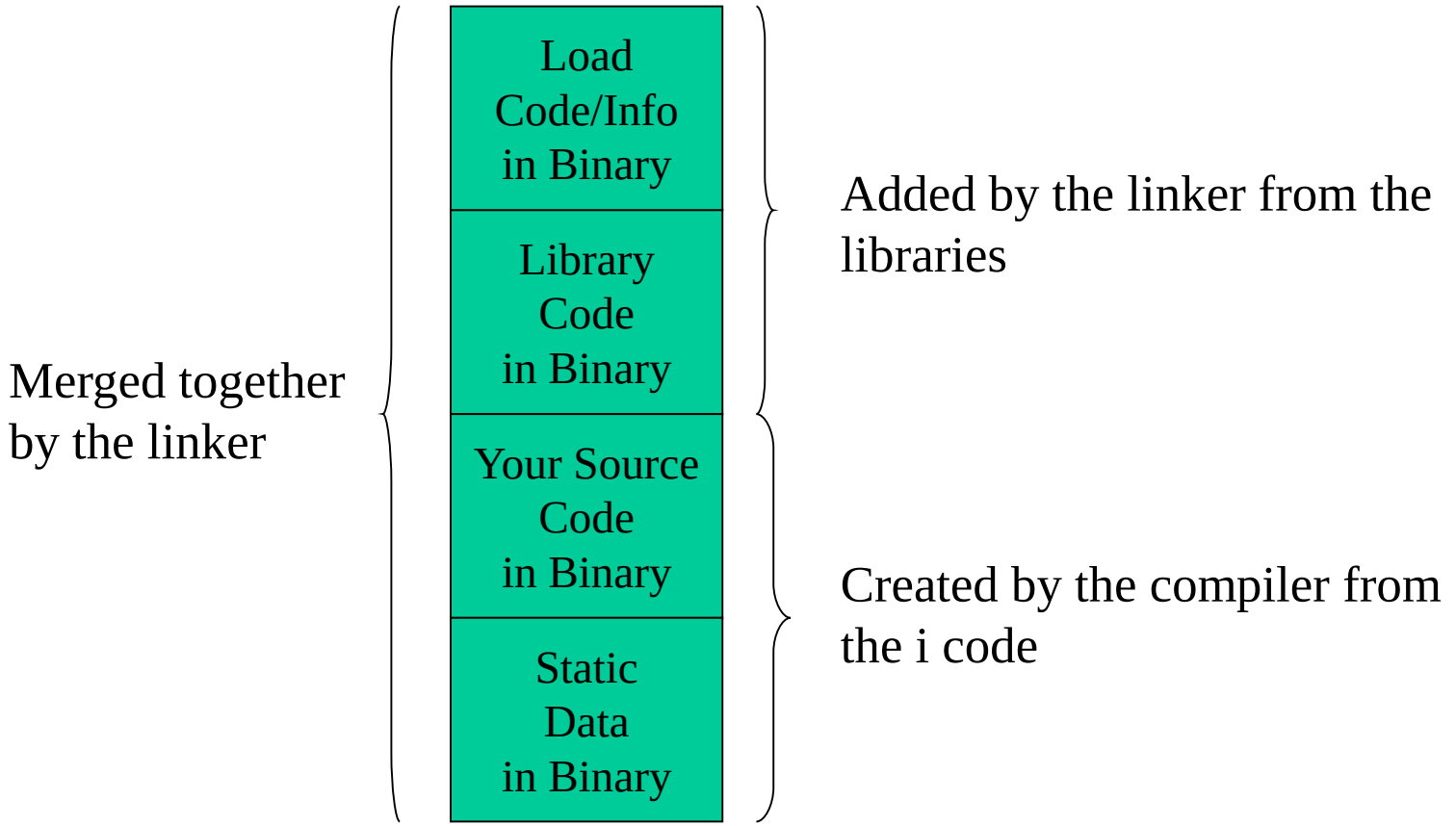


Note: The compiler does not compile the Text File you entered but the i File, it has been changed by the Pre-Processor.

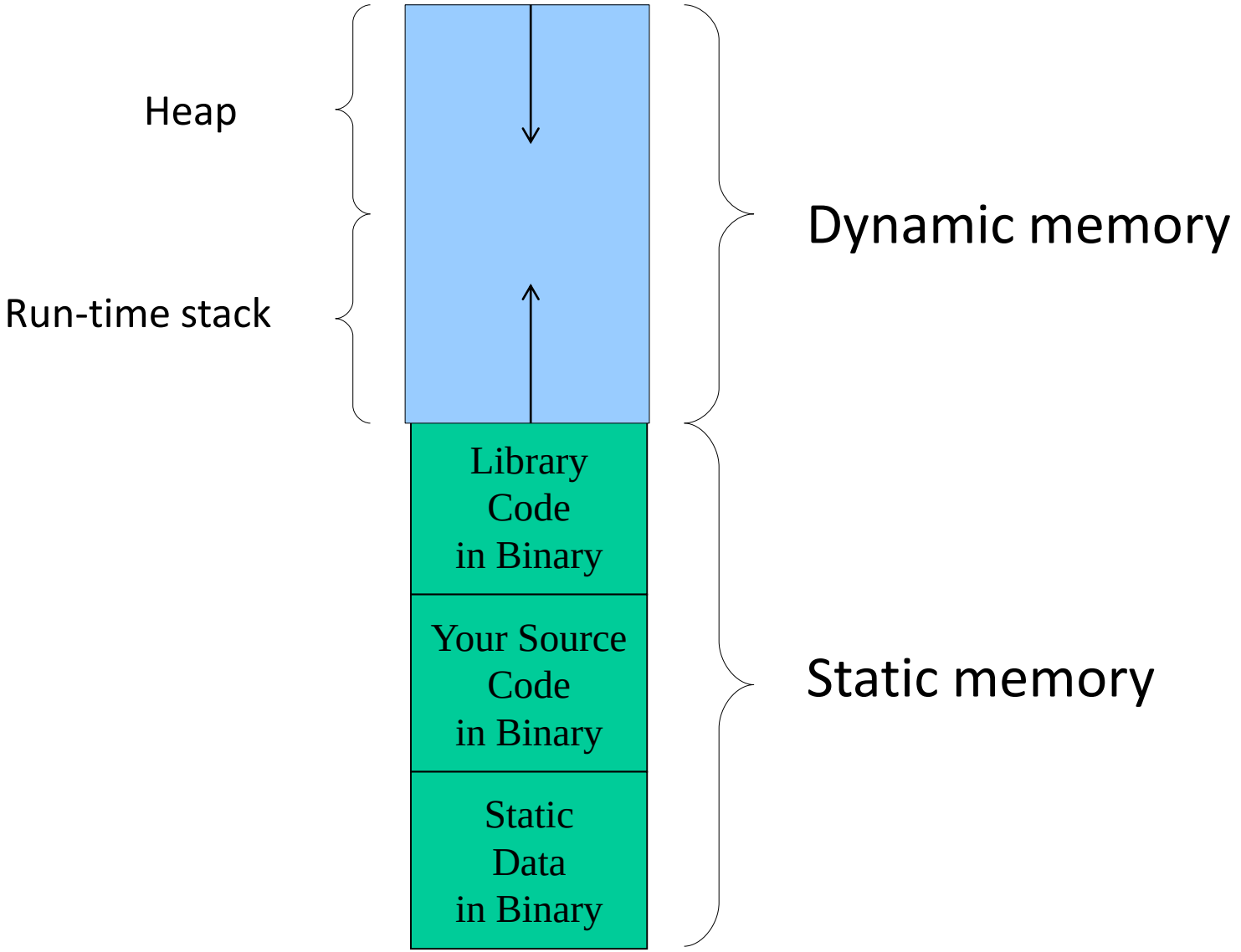
C Files

- Source Files
  - FILENAME.c .... the program
  - FILENAME.h .... header file (shared code)
- Pre-processed File
  - FILENAME.i
- Object Files and Assembler Files
  - FILENAME.o
  - FILENAME.s
- Executable Files
  - FILENAME .... Using the –o switch
  - a.out .... the default executable name

Structure of a Compiled File



Structure of a Process



The pre-processor

More on this later, but…

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

Is a pre-processor command.

Week 5 Lecture 3

C Control Structures & Variables

Types & Variables

Built-in C Language Types

DESCRIPTION	RESERVED WORD	BITS	RANGE
Integer	short	8	- 128 to + 127
	int	16	+/- 32,768
	long	32	+/- 2,147,483,648
Floating Point	float	32	+/- 3.4 x 10^{38} with 7 significant digits
	double	64	+/- 1.7 x 10^{308} with 15 significant digits
Boolean	short, int, long	(0 is false, other true)	
Character	char, unsigned short int	8	0 to 256
String	char *	32	address in memory (special case of pointer)
Pointers	TYPE*	32	address in memory

Bit sizes depend on CPU

Variable Declaration

Syntax:

```
SCOPE   MODIFIER   TYPE  VAR_NAME;
SCOPE   MODIFIER   TYPE  VAR_NAME = VALUE;
SCOPE   MODIFIER   TYPE  VAR1, VAR2, … , VARn;
```

Where:

SCOPE	- static, extern or it is not used
MODIFIER	- unsigned,short, long or not used
TYPE	- one of the built-in types
VAR_NAME	- must start with a character, any word not beginning with a number or a reserved symbol (like +, -). It is case sensitive: for, For, fOr

Variable Declaration

- int x;
- int x, y, z;
- int x = 5, y, z = 2;
- short int a = -2;
- unsigned short int b = 4;
- char c = 4;

Constant Declarations

- In C, a variable can be declared as constant.
- The value of a constant is initialized when the variable is declared. That value cannot be changed.

```
int const a = 1;
const int a = 2;
```

typedef Declaration

The typedef command allows for the creation of custom type names. This makes the program more readable.

```
typedef int scalefactor;   // a simple example

int main() {

    scalefactor a;

    a = 10;
    printf("The scale factor is:%d", a);
}
```

typedef Declaration

```
typedef int boolean;
int true=1, false=0;

boolean isValid = false;
```

Operators & Expressions

Operators

Math

Assuming integer math

+	add	x = 5 + 2;	// x becomes 7
-	subtract	x = 5 – 2;	// x becomes 3
*	multiply	x = 5 * 2;	// x becomes 10
/	divide	x = 5 / 2;	// x becomes 2
%	modulo	x = 5 % 2;	// x becomes 1
++	increment	x = x++;	// if x=5 then x=6
--	decrement	x = x--;	// if x=5 then x=4
+=	increment by	x += 3;	// if x=5 then x=8
-=	decrement by	x -= 3;	// if x=5 then x=2
*=	multiply by	x *= 3;	// if x=5 then x=15
/=	divide by	x /= 3;	// if x=5 then x=1

Operators

Logical

• <	less than	5 < 10	true
• >	greater than	5 > 10	false
• <=	less and equal	5 <= 5	true
• >=	greater and equal	5 >= 5	true
• ==	equal	5 == 10	false
• !=	not equal	5 != 10	true
• !	Not	! (5 == 10)	true
• &&	and	(5<10)&&(5>10)	false
• ||	or	(5<10)||(5>10)	true

Expressions

With assignment

- VARIABLE = EXPRESSION;
  - x = 5 + y; // x will contain 5 more than y
  - x = 5 > 10; // results in 1 for true, or 0 for false

Without assignment

- (EXPRESSION)
  - if (x < 10) // true when x is less than 10
  - if (x + 2) // true when result is not equal to zero

Notice the low-level features of C, where logical expressions and integer mathematics mix.

Complex expressions

What do the following output?

- x = x + y++; // assume x = 5 and y = 2
- x = x + ++y;

Standard C definition:
- Var++ → increment after solving the expression
- ++Var → increment before solving the expression

- VAR = (CONDITION) ? TRUE_EXPRESSION : FALSE_EXPRESSION;

- x = (y < 10) ? x++ : x--; // assume x = 5 and y = 2

Control Structures

The if-statement

```
if (CONDITION) SINGLE_STATEMENT;
    if (x < 10) puts("X is less than 10\n");

if (CONDITION) { MULTIPLE_STATEMENTS; }
    if (x < 10) {
        puts("X is less than 10\n");
        c = getc(stdin);
    }
```

The if-statement

```
if (CONDITION) SINGLE_STATEMENT; else SINGLE_STATEMENT;
    if (x < 10) puts("X is less than 10\n");
    else puts("X is greater than 10\n");

if (CONDITION) { MULTIPLE_STATEMENTS; }
else {MULTIPLE_STATEMENTS;}
    if (x < 10) {
        puts("X is less than 10\n");
        c = getc(stdin);
    } else {
        puts("X is greater than 10\n");
    }
```

The switch-statement

```
switch(VARIABLE) {
   case VALUE:     MULTIPLE_STATEMENTS;
                   break;
   case VALUE2:    MULTIPLE_STATEMENTS;
                   break;
   default:
        MULTIPLE_STATEMENTS;
}
```

Optional. It designates the end of the case block. If not present executions automatically goes to the next case block without testing the condition.

The VALUE is a constant, it cannot be a variable.

The VARIABLE can only be of type integer or character.

Some new compilers permit strings.

The switch-statement

```
switch(age) {
    case 1:  puts("You are too young, sorry!\n");
             break;
    case 2:
    case 3: puts("You must attend with a parent.\n");
             break;
    default:
          puts("You can drive the car.\n");
}
```

We are assuming that the variable AGE is an integer.
If the value is a 1 then the first case is activated only.
If the value is a 2 or 3 then the second case is activated.
All other integer values are handled by the default case.

The switch-statement

```
switch(gender) {
    case ‘m’:
    case ‘M’:    puts(“Girls are only welcome.\n”);
                 break;
    case ‘f’:
    case ‘F’:    puts(“Welcome.\n”);
                 break;
    default:
         puts(“Please enter an F or an M.\n”);
}
```

The while-loop

```
while (CONDITION_IS TRUE) SINGLE_STATEMENT;

while (CONDITION_IS_TRUE) { MULTIPLE_STATEMENTS; }
```

```
int x = 0;
while (x < 10) x++;
```

```
int x = 10;
while(x--);
```

```
int y = 20;
while (y > 0) {
    puts("Hi!");
    y--;
}
```

The do-while-loop

```
do SINGLE_STATEMENT; while (CONDITION_IS_TRUE);

do { MULTIPLE_STATEMENTS; } while (CONDITION_IS_TRUE);

char gender;
do {
    puts("Gender (M or F): ");
    gender = getc(stdin);
} while (gender!='M' && gender!='F');
```

The for-loop

for (START; CONDITION; EXPRESSION) SINGLE_STATEMENT;

for (START; CONDITION; EXPRESSION) { MULTIPLE_STATEMENTS; }

```
int x;
for(x=0; x<10; x++) puts("Hi!");
```

```
int x, y;
char c;
for(x=0, y=10, c=' '; x<10 && c != 'x'; x+=2, y--) {
    puts("Hi");
    c = getc(stdin);
}
```

START and EXPRESSION are comma-separated lists.

START, CONDITION, and EXPRESSION are optional !!

```
for(;;);

 for(;x<10;) x--;
```