

CS 2211

Systems Programming

Part Two: C Fundamentals

Program: Printing a Pun

```
#include <stdio.h>

int main(void)
{
    printf("To C, or not to C: that is the question.\n");
    return 0;
}
```

This program might be stored in a file named `pun.c`.

The file name doesn't matter, but the `.c` extension is often required.

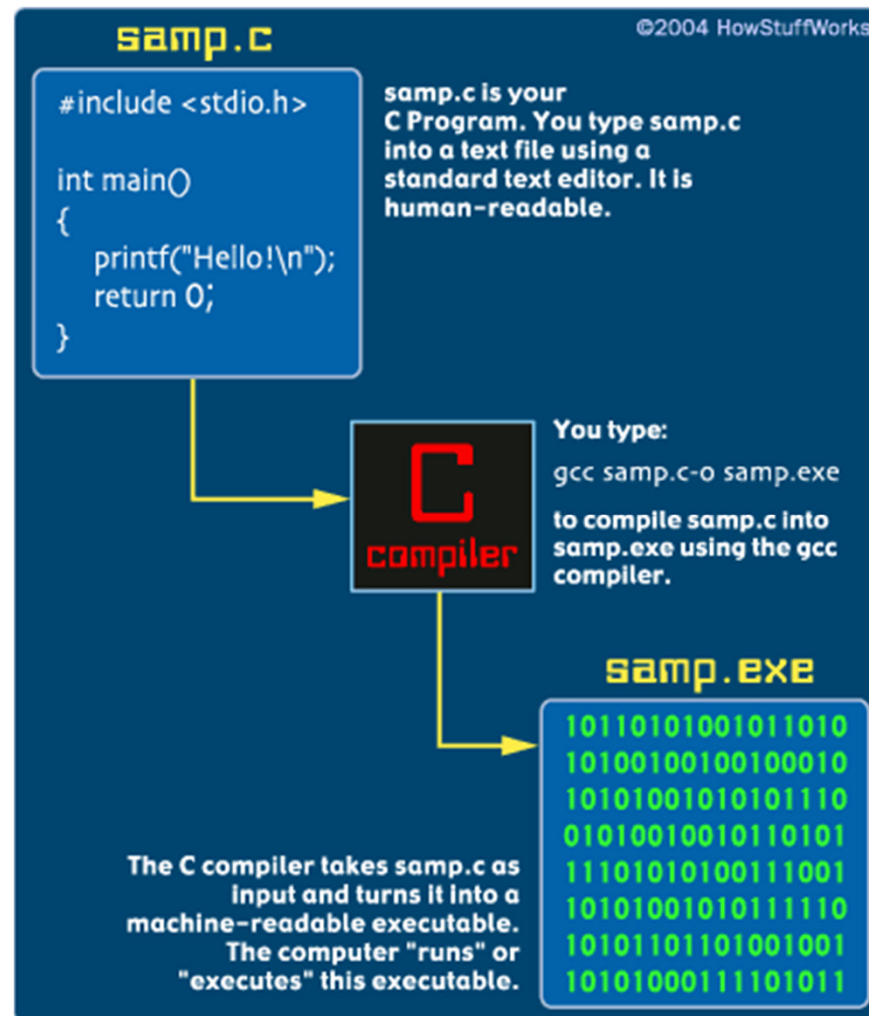
Compiling and Linking

Before a program can be executed, three steps are usually necessary:

- (1) **Preprocessing.** The *preprocessor* obeys commands that begin with # (known as *directives*)
- (2) **Compiling.** A *compiler* translates then translates the program into machine instructions (*object code*).
- (3) **Linking.** A *linker* combines the object code produced by the compiler with any additional code needed to yield a complete executable program.

The preprocessor is usually integrated with the compiler.

COMPILING a C PROGRAM



Compiling and Linking Using `cc`

To compile and link the `pun.c` program under UNIX, enter the following command in a terminal or command-line window:

```
% cc pun.c
```

The `%` character is the UNIX prompt.

Linking is automatic when using `cc`; no separate link command is necessary.

Compiling and Linking Using `cc`

After compiling and linking the program, `cc` leaves the executable program in a file named `a.out` by default.

The `-o` option lets us choose the name of the file containing the executable program.

The following command causes the executable version of `pun.c` to be named `pun`:

```
% cc -o pun pun.c
```

The GCC Compiler

GCC is one of the most popular C compilers.

GCC is supplied with Linux but is available for many other platforms as well. Using this compiler is similar to using **cc**:

```
% gcc -o pun pun.c
```

The General Form of a Simple Program

- Simple C programs have the form

directives

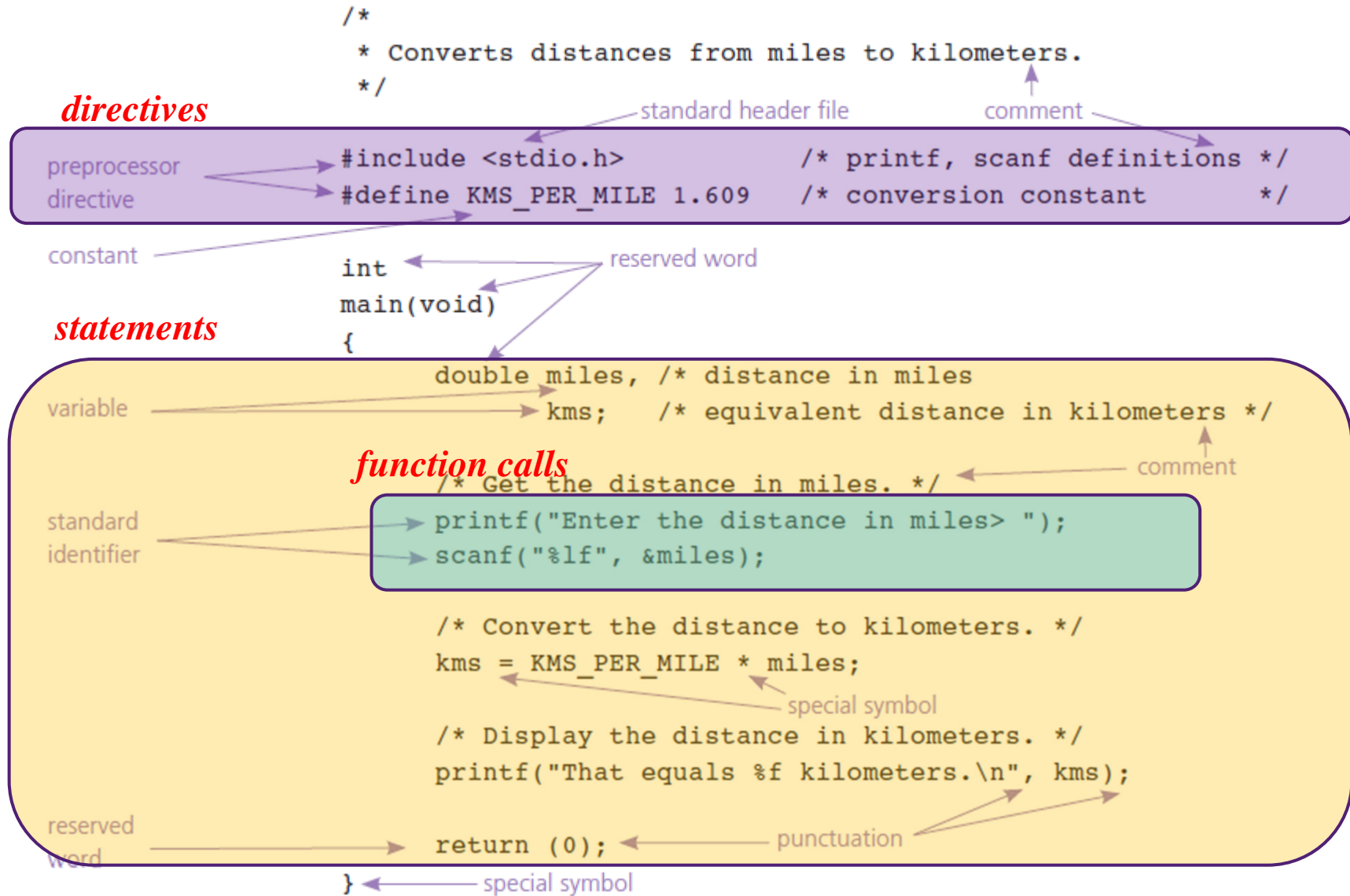
```
int main(void)
```

```
{
```

```
    statements
```

```
}
```


ANATOMY OF A C PROGRAM



C Fundamentals

END OF PART 1 - A

The General Form of a Simple Program

- C uses { and } in much the same way that some other languages use words like **begin** and **end**.

{ - begin

} - end

The General Form of a Simple Program

- Even the simplest C programs rely on three key language features:
 - **Directives**
 - **Functions**
 - **Statements**

Directives

- Directives always begin with a # character.
- By default, directives are one line long;
 - there's no semicolon
or other special marker at the end.

Directives

- Before a C program is compiled, it is first edited by a preprocessor.
- Commands intended for the preprocessor are called directives.
- Example:
`#include <stdio.h>`
- `<stdio.h>` is a *header* containing information about C's standard I/O library.

Functions

- A *function* is a series of statements that have been grouped together and given a name.
- *Library functions* are provided as part of the C implementation.
- A function that computes a value uses a **return** statement to specify what value it “returns”:

```
return x + 1;
```

Functions

- *Library functions* are provided as part of the C implementation.
- Example:
`#include <stdio.h>`
- `<stdio.h>` is a *header* containing information about C's standard I/O library.

```
printf( "2nd value of b is : %d \n" , b );
```

```
scanf ("%d", %b);
```


The `main` Function

- The `main` function is mandatory.
- `main` is special: it gets called automatically when the program is executed.
- `main` returns a status code; the value `0` indicates normal program termination.
- If there's no `return` statement at the end of the `main` function, many compilers will produce a warning message.

Statements

- A *statement* is a command to be executed when the program runs.
- **pun.c** uses only two kinds of statements. One is the `return` statement; the other is the *function call*.
- Asking a function to perform its assigned task is known as *calling* the function.
- `pun.c` calls **printf** to display a string:

```
printf("To C, or not to C: that is the question.\n");
```

Statements

- C requires that each statement end with a **semicolon**.
- **Directives** are normally one line long, and they don't end with a semicolon.
- Example:
#include <stdio.h>

Comments

A *comment* begins with `/*` and end with `*/`.

```
/* This is a comment */
```

Comments may appear almost anywhere in a program, either on separate lines or on the same lines as other program text.

Comments may extend over more than one line.

```
/* Name: pun.c  
   Purpose: Prints a bad pun.  
   Date: July 06, 2042*/
```

Comments

Warning: Forgetting to terminate a comment may cause the compiler to ignore part of your program:

```
printf("My ");    /* forgot to close this comment...  
printf("cat ");  
printf("has ");   /* so it ends here */  
printf("fleas");
```

Comments in C99

In C99, comments can also be written in the following way:

```
// This is a comment
```

This style of comment ends automatically at the end of a line.
Advantages of **//** comments:

Safer: there's no chance that an unterminated comment will accidentally consume part of a program.

Multiline comments stand out better.

C Fundamentals

END OF PART 1 - B

Variables in C

Most programs need to a way to store data temporarily in computer memory during program execution.

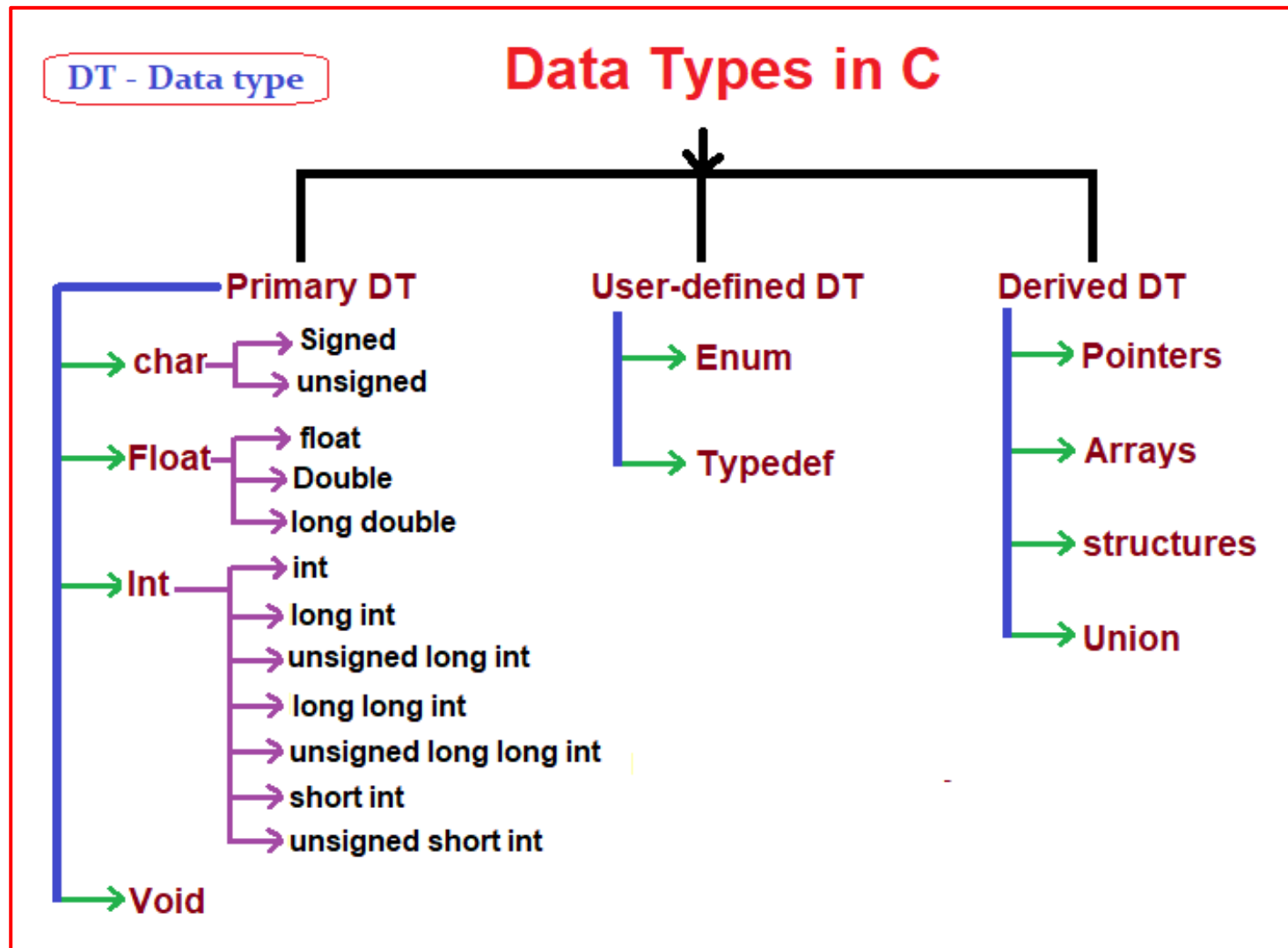
These storage locations are called *variables*.

Types

Every variable must have a *type*.

C has a wide variety of types, including `int` and `float`.

A variable of type `int` (short for *integer*) can store a whole number such as 0, 1, 392, or -2553.

BREAKDOWN of DATA TYPES

C's built-in data types that are similar to ones in Java:

Syntax	Name	Java's counterpart	use
<code>char</code>	character	<code>byte</code>	Stores an ASCII code (character) or it can also store a very short integer (−128..127)
<code>short</code>	short integer	<code>short</code>	uses 2 byte memory, value between −32768 and 32767
<code>int</code>	ordinary integer	<code>int</code>	uses 4 byte memory, value between −2147483648 and 2147483647
<code>long</code>	long integer	<code>long</code>	uses 8 bytes memory, value between −9223372036854775808 and 9223372036854775807
<code>float</code>	single precision float	<code>float</code>	uses 4 byte memory, absolute value between 1.4E−45 and 3.4E38
<code>double</code>	double precision float	<code>double</code>	uses 8 byte memory, absolute value between 4.9E−324 and 1.8E308
<code>_Bool</code>	<code>boolean</code>	<code>boolean</code>	true (1) or false (0)

C's built-in data types that do not have an equivalent in Java:

Syntax	Name	use
<code>unsigned char</code>	Unsigned character	Very short positive integer (0..255)
<code>unsigned short</code>	Unsigned short integer	uses 2 byte memory, value between 0 and 65535
<code>unsigned int</code>	Unsigned ordinary integer	uses 4 byte memory, value between 0 and 4294967295
<code>unsigned long</code>	Unsigned long integer	uses 8 bytes memory, value between 0 and 18446744073709551615
<code>*</code>	<i>Reference type</i>	Contains a memory address (usually 4 bytes, but 64 bits machines will use 8 bytes)

Declarations

Variables must be *declared* before they are used.

Variables can be declared one at a time:

```
int height;  
float profit;
```

Alternatively, several can be declared at the same time:

```
int height, length, width, volume;  
float profit, loss;
```

Declarations

When **main** contains declarations, these must precede statements:

```
int main(void)
{
    declarations
    statements
}
```

In C99, declarations don't have to come before any statements.

(you can have a series of statements and then some variable declarations followed by other statements)

C Fundamentals

END OF PART 1 - C

Variable Initialization [definition]

Some variables are automatically set to zero when a program begins to execute, but most are not.

A variable that doesn't have a default value and hasn't yet been assigned a value by the program is said to be *uninitialized*.

Attempting to access the value of an uninitialized variable may yield an unpredictable result.

With some compilers, worse behavior—even a program crash—may occur.

Initialization

The initial value of a variable may be included in its declaration:

```
int height = 8;
```

The value **8** is said to be an *initializer*.

Any number of variables can be initialized in the same declaration:

```
int height = 8, length = 12, width = 10;
```

Each variable requires its own initializer.

```
int height, length, width = 10;  
/* initializes only width */
```


Assignment

A variable can be given a value by means of *assignment*:

```
height = 8;
```

The number **8** is said to be a *constant*.

Before a variable can be assigned a value—or used in any other way—it must first be **declared**.

Declared: (variable is given a type and a name [identifier])

- versus -

Defined: (variable is set with a value)

Assignment

A constant assigned to a **float** variable usually contains a decimal point:

```
profit = 2150.48;
```

It's best to append the letter **f** to a floating-point constant if it is assigned to a **float** variable:

```
profit = 2150.48f;
```

Failing to include the **f** may cause a warning from the compiler.

Assignment

Once a variable has been assigned a value, it can be used to help compute the value of another variable:

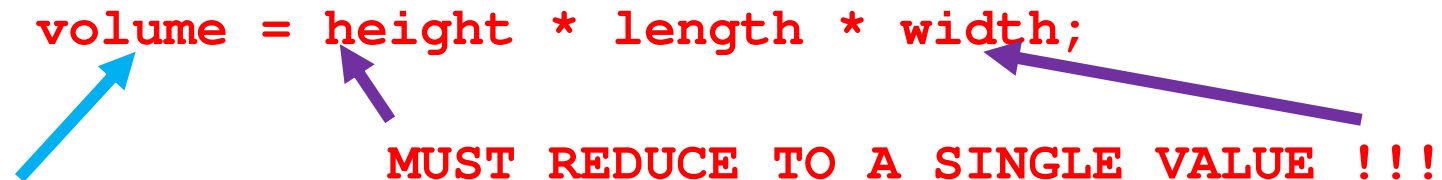
```
height = 8;  
length = 12;  
width = 10;  
volume = height * length * width;  
/* volume is now 960 */
```

Assignment

VERY IMPORTANT:

= (this is the assignment statement !)
i.e. 'assigned' and NOT 'equal to'


height = 8;
LHS RHS


volume = height * length * width;
MUST REDUCE TO A SINGLE VALUE !!!

Resolves to a
specific location
in memory

Defining Names for Constants

Assume you require a constant value of 166 on your program, whose meaning may not be clear to someone reading the program.

Using a feature known as *macro definition*, we can name this constant:

```
#define INCHES_PER_POUND 166
```

Defining Names for Constants

When a program is compiled, the preprocessor replaces each macro by the value that it represents.

During preprocessing, the statement

```
weight = (volume + INCHES_PER_POUND - 1) / INCHES_PER_POUND;
```

will become

```
weight = (volume + 166 - 1) / 166;
```

Defining Names for Constants

The value of a macro can be an expression:

```
#define RECIPROCAL_OF_PI (1.0f / 3.14159f)
```

If it contains operators, the expression should be enclosed in parentheses.

Using only upper-case letters in macro names is a common convention.

Identifiers

Names for variables, functions, macros, and other entities are called *identifiers*.

An identifier may contain letters, digits, and underscores, but must begin with a letter or underscore:

`times10` `get_next_char` `_done`

It's usually best to avoid identifiers that begin with an underscore.

Examples of illegal identifiers:

`10times` `get-next-char`

Identifiers

C is *case-sensitive*: it distinguishes between upper-case and lower-case letters in identifiers.

For example, the following identifiers are all different:

`job` `joB` `jOb` `jOB` `Job` `JoB` `JOB` `JOB`

Identifiers

Many programmers use only lower-case letters in identifiers (other than macros), with underscores inserted for legibility:

symbol_table current_page name_and_address

Other programmers use an upper-case letter to begin each word within an identifier:

symbolTable currentPage nameAndAddress

C places no limit on the maximum length of an identifier.

Keywords

The following *keywords* can't be used as identifiers:

auto	enum	restrict*	unsigned
break	extern	return	void
case	float	short	volatile
char	for	signed	while
const	goto	sizeof	_Bool*
continue	if	static	_Complex*
default	inline*	struct	_Imaginary*
do	int	switch	
double	long	typedef	
else	register	union	

*C99 only

Keywords

Keywords (with the exception of

`_Bool`

`_Complex`

`_Imaginary`)

must be written using only lower-case letters.

Names of **library functions** (e.g., **`printf`**) are also lower-case.

Layout of a C Program

The amount of space between tokens usually isn't critical. At one extreme, tokens can be crammed together with no space between them, except where this would cause two tokens to merge:

```
/* Converts a Fahrenheit temperature to Celsius */  
#include <stdio.h>  
#define FREEZING_PT 32.0f  
#define SCALE_FACTOR (5.0f/9.0f)  
int main(void){float fahrenheit,celsius;printf(  
"Enter Fahrenheit temperature: ");scanf("%f", &fahrenheit);  
celsius=(fahrenheit-FREEZING_PT)*SCALE_FACTOR;  
printf("Celsius equivalent: %.1f\n", celsius);return 0;}
```

Layout of a C Program

The whole program can't be put on one line, because each preprocessing directive requires a separate line.

Compressing programs in this fashion isn't a good idea.

In fact, adding spaces and blank lines to a program can make it easier to read and understand.

Layout of a C Program

C allows any amount of space—blanks, tabs, and new-line characters—between tokens.

Consequences for program layout:

Statements can be divided over any number of lines.

Space between tokens (such as before and after each operator, and after each comma) makes it easier for the eye to separate them.

Indentation can make nesting easier to spot.

Blank lines can divide a program into logical units.

Layout of a C Program

Although extra spaces can be added between tokens, it's not possible to add space within a token without changing the meaning of the program or causing an error.

Writing

```
fl oat fahrenheit, celsius;  /*** WRONG ***/
```

or

```
fl  
oat fahrenheit, celsius;      /*** WRONG ***/
```

produces an error when the program is compiled.

Layout of a C Program

Putting a space inside a string literal is allowed, although it changes the meaning of the string.

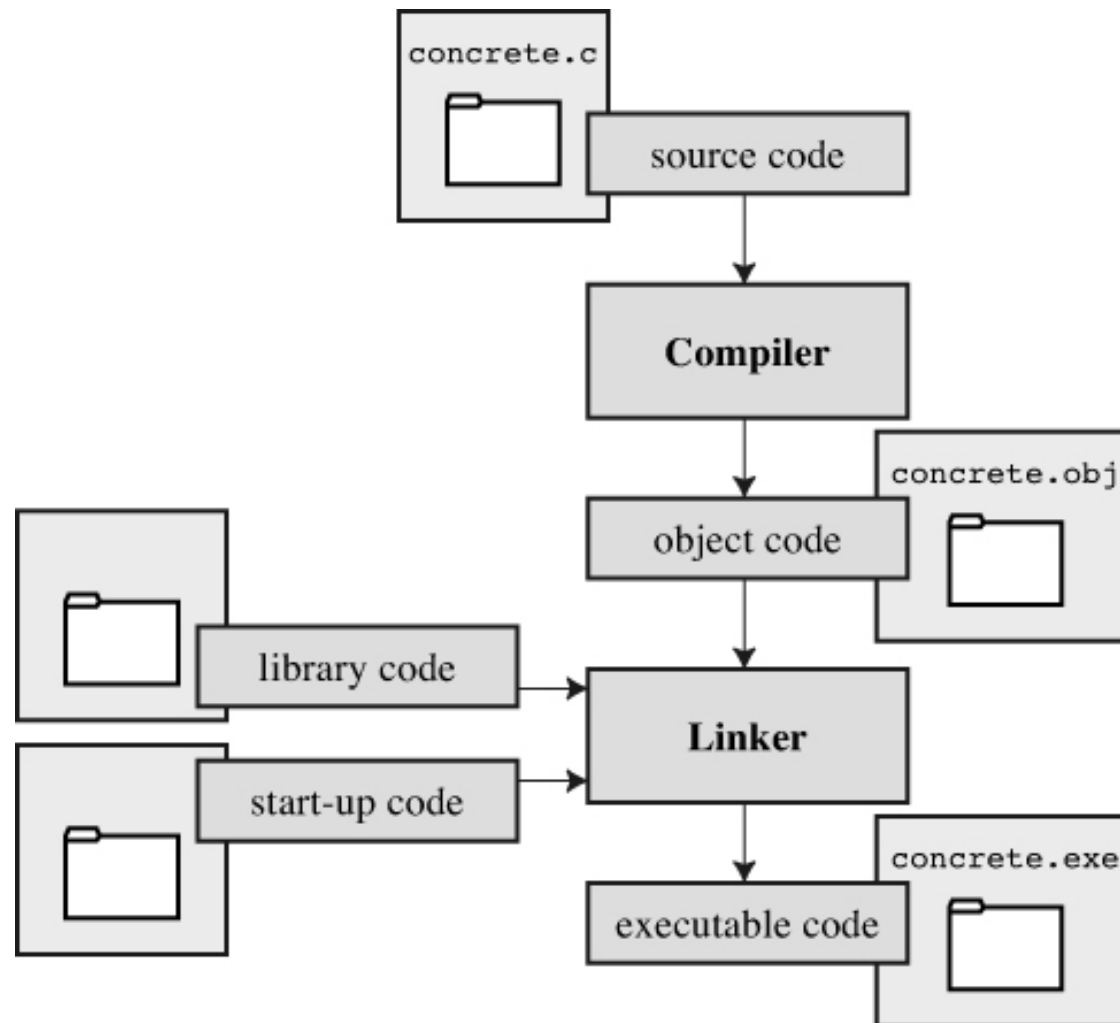
Putting a new-line character in a string (splitting the string over two lines) is illegal:

```
printf("To C, or not to C:  
that is the question.\n");  
/*** WRONG ***/
```

C Fundamentals

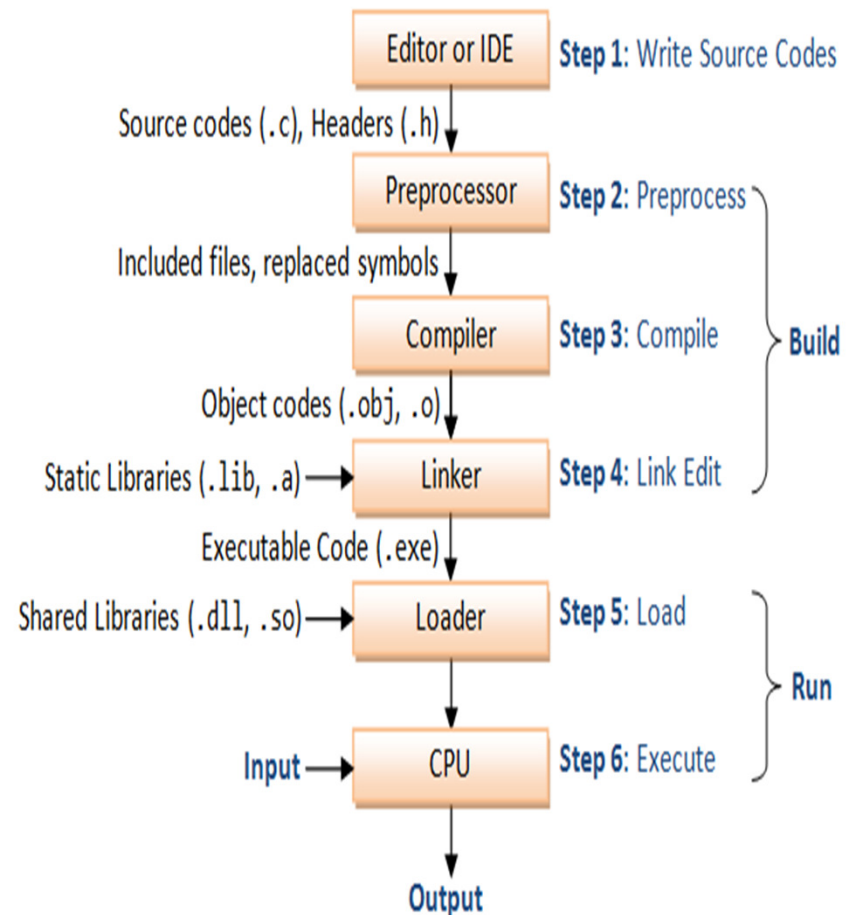
END OF PART 2 - A

COMPILING a C PROGRAM



COMPILING a C PROGRAM

- Step 1:** Write the source codes (.c) and header files (.h).
- Step 2:** Pre-process the source codes according to the preprocessor directives. The preprocessor directives begin with a hash sign (#).
- Step 3:** Compile the pre-processed source codes into object codes (.obj, .o).
- Step 4:** Link the compiled object codes with other object codes and the library object codes (.lib, .a) to produce the executable code (.exe).
- Step 5:** Load the executable code into computer memory.
- Step 6:** Run the executable code.



COMPILING a C PROGRAM

Step 2: Pre-process the source codes according to the preprocessor directives.

Before invoking the C compiler, the C programming language system will always invoke a **C pre-processor** to process the program source code.

Tasks performed by the C pre-processor:

Removes comments from the source code

```
/* ..... */  
or: // .....
```

Read in included file

```
#include <stdio.h>  
or #include "header.h"
```

Process macro (symbolic) definitions

```
#define ... ..
```

Other advanced conditionals:

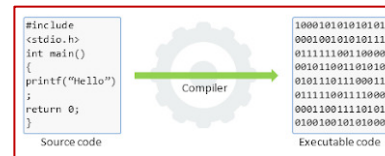
```
#ifdef ...  
...  
#endif
```

```
#ifndef ...  
...  
#endif
```

COMPILING a C PROGRAM

Step 3: Compile the pre-processed source codes into object codes (.obj, .o).

A object file (.o) contains machine instructions in binary code
It's not for human consumption !



```
int f( int x )
{
    return ( x*x );
}
```

Compiled code:

0000000000000000 <f>:

0: 0011010101101011

1: 1101010101000110

4: 1001011010100011

7: 1011110101001000

a: 0011010110110011

e: 0101010110001101

f: 11101110110101100

push %rbp

mov %rsp,%rbp

mov %edi,-0x4(%rbp)

mov -0x4(%rbp),%eax // get x in reg. eax

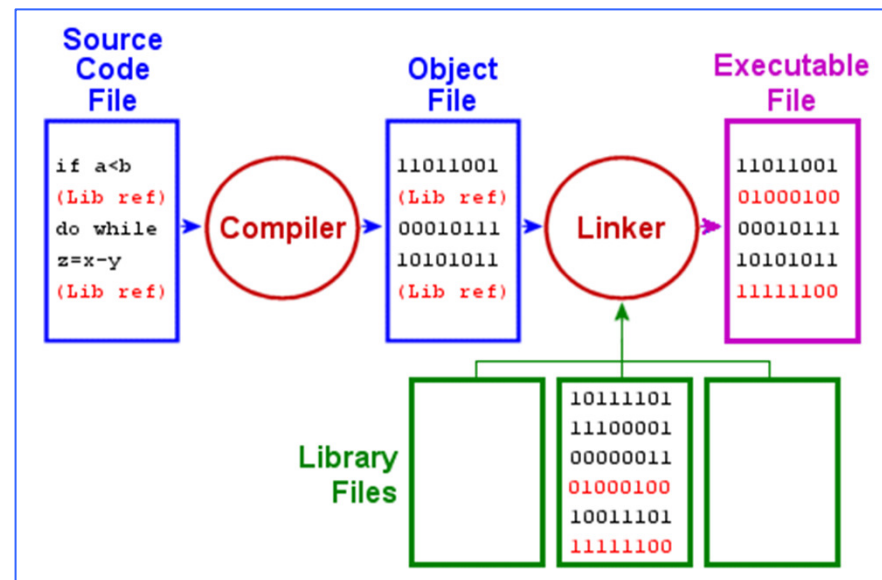
imul -0x4(%rbp),%eax // (x*x)

leaveq

retq

COMPILING a C PROGRAM

Step 4: Link the compiled object codes with other object codes and the library object codes (.lib, .a) to produce the executable code (.exe).



The linker will **delineate (assign) memory space** for every variable

So: the variables z, x and y will be stored somewhere.

Depending on where the variables are stored,
the linker will patch the relocation location with the allocated location

C Fundamentals

END OF PART 3 - A