

A Brief Tour

symbol key

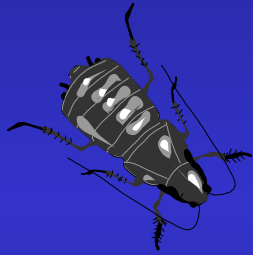
2



compiles



doesn't
compile



a bug



a note



advanced - don't
use!
(yet)



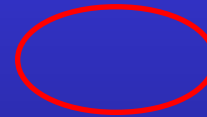
we're happy



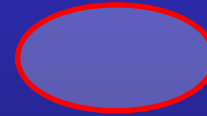
it depends



we're not
happy



highlight



key to the
nature of C

c99

not in c89

- the traditional first program

preprocessor directives to include header files

program execution starts with a function called main

string literals are double-quote delimited

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

int main(void)
{
    puts("Hello, world");
    return EXIT_SUCCESS;
}
```

simple statements are semi-colon terminated

function definitions are curly-brace delimited

The diagram shows a C program with several annotations. Red arrows point from text labels to specific parts of the code. The labels are: 'preprocessor directives to include header files' pointing to the two #include lines; 'program execution starts with a function called main' pointing to the main function signature; 'string literals are double-quote delimited' pointing to the string in the puts call; 'simple statements are semi-colon terminated' pointing to the two lines of code inside the function body; and 'function definitions are curly-brace delimited' pointing to the opening and closing braces of the main function.

- there are 5 kinds of tokens in C programs

tokens

<i>token</i>	<i>examples</i>
identifiers	hiker main result
keywords	if struct restrict
literals	9 6 13 '9' 9.6
operators	+ - * / % =
punctuators	; { } ,

- **C is a free-form language**
 - ◆ whitespace and comments can always separate program tokens
- **multi-line comments**
 - ◆ start with `/*` and end with `*/`
- **single-line comments**
 - ◆ start with `//`

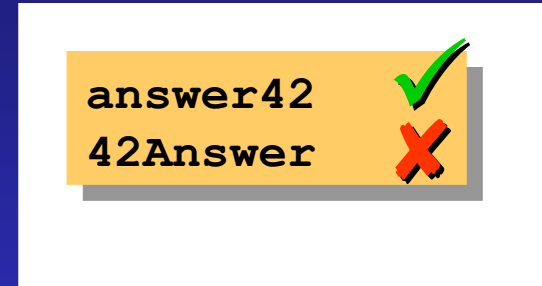
multi-line
comment

```
#include <stdio.h>    /*  
                    * puts  
                    */  
#include <stdlib.h>  // EXIT_SUCCESS
```

single-line comment

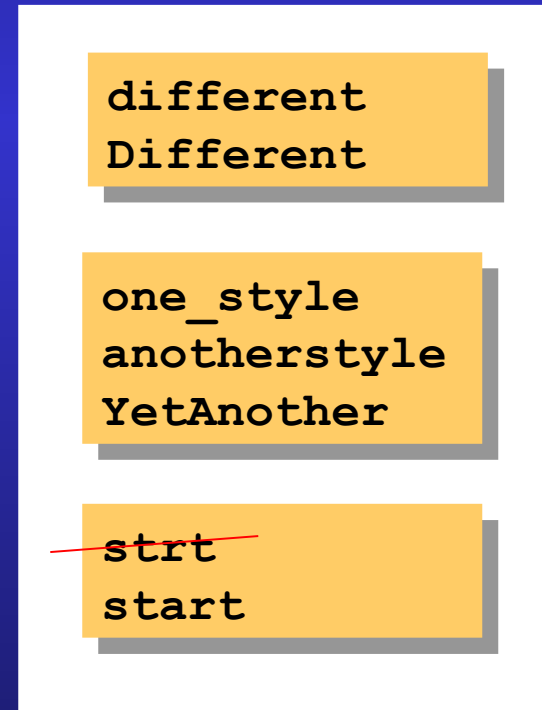
- **rules**

- ◆ made of letters, digits and underscores
- ◆ can't start with a digit
- ◆ case sensitive



- **recommendations**

- ◆ don't start with underscore (reserved)
- ◆ use case consistently
- ◆ avoid abbrs
- ◆ don't use clever spelling
- ◆ don't use hungarian



- **some identifiers have a fixed meaning**
 - ◆ the easy part of learning the language is learning these fixed meanings
 - ◆ be careful – some keywords have more than one meaning depending on the context

```
#include <stdio.h>    /* printf */  
  
int main(void)  
{  
    printf("Hello, world\n");  
    return 0;  
}
```

- **37 keywords**
 - ◆ `<stdbool.h>` provides a typedef for `bool` and macros for `true` and `false`
 - ◆ `wchar_t` is a typedef not a keyword
 - ◆ worth also avoiding C++ keywords

new in C99

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	



- some tokens represent specific values

<i>type</i>	<i>literal</i>	
bool	true false	truth
double	3.1415	floating point
int	9 6 13 42	integers
char	'X'	characters
char *	"Hello"	sequences of characters

- some tokens represent specific actions

<i>symbol</i>	<i>category</i>
+ - * / %	arithmetic
== !=	(in)equality
< <= > >=	relational
&& ! ?:	logical
= += -= *= /=	assignment
& ^ ~ << >>	bitwise

- some tokens group or separate other tokens
 - ◆ { and } form blocks of statements
 - ◆ semi-colons mark the end of a simple statement

```
#include <stdio.h>    /* printf, putchar */  
#include <stdlib.h>  /* EXIT_SUCCESS */
```

```
int main(void)
```

```
{
```

```
    int result = 9 * 6;
```

```
    int thirteen = 13;
```

```
    printf("%i", result / thirteen);
```

```
    printf("%i", result % thirteen);
```

```
    putchar('\n');
```

```
    return EXIT_SUCCESS;
```

```
}
```

- **declarations introduce variables**
 - ◆ a variable has an identifier and a type
 - ◆ a variable's type can never change

```
type identifier ;
```

```
type identifier, another ;
```

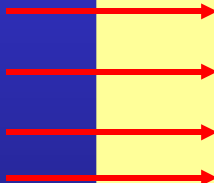
```
#include <stdio.h>    /* printf, putchar */

int main(void)
{
    int result = 9 * 6;
    int thirteen = 13;
    printf("%d", result / thirteen);
    printf("%d", result % thirteen);
    putchar('\n');
    return 0;
}
```

- **expressions compute things!**
 - ◆ an expression yields a value
 - ◆ an expression may or may not have a side-effect

```
#include <stdio.h>    /* printf, putchar */

int main(void)
{
    int result = 9 * 6;
    int thirteen = 13;
    printf("%d", result / thirteen);
    printf("%d", result % thirteen);
    putchar('\n');
    return 0;
}
```



- a program to reverse the command line
 - ◆ typical of the terseness found in C programs

argc is the argument
count of argv

argv is an array of
strings

```
#include <stdio.h>    /* printf, putchar */  
  
int main(int argc, char * argv[])  
{  
    while (argc > 0)  
        printf("%s ", argv[--argc]);  
    putchar('\n');  
  
    return 0;  
}
```

this is an alternative signature for
main

example

- **evaluation order is not always determined by the layout order of the code**
 - ◆ **statements, initialisers, operands of the short-circuiting and comma operators are evaluated in strict sequence... and that's pretty much it**
 - ◆ **can affect expression certainty and correctness**

```
a = f() * g() + h();  
a = f() * (g() + h());
```

what is the difference in evaluation order between these two assignments?

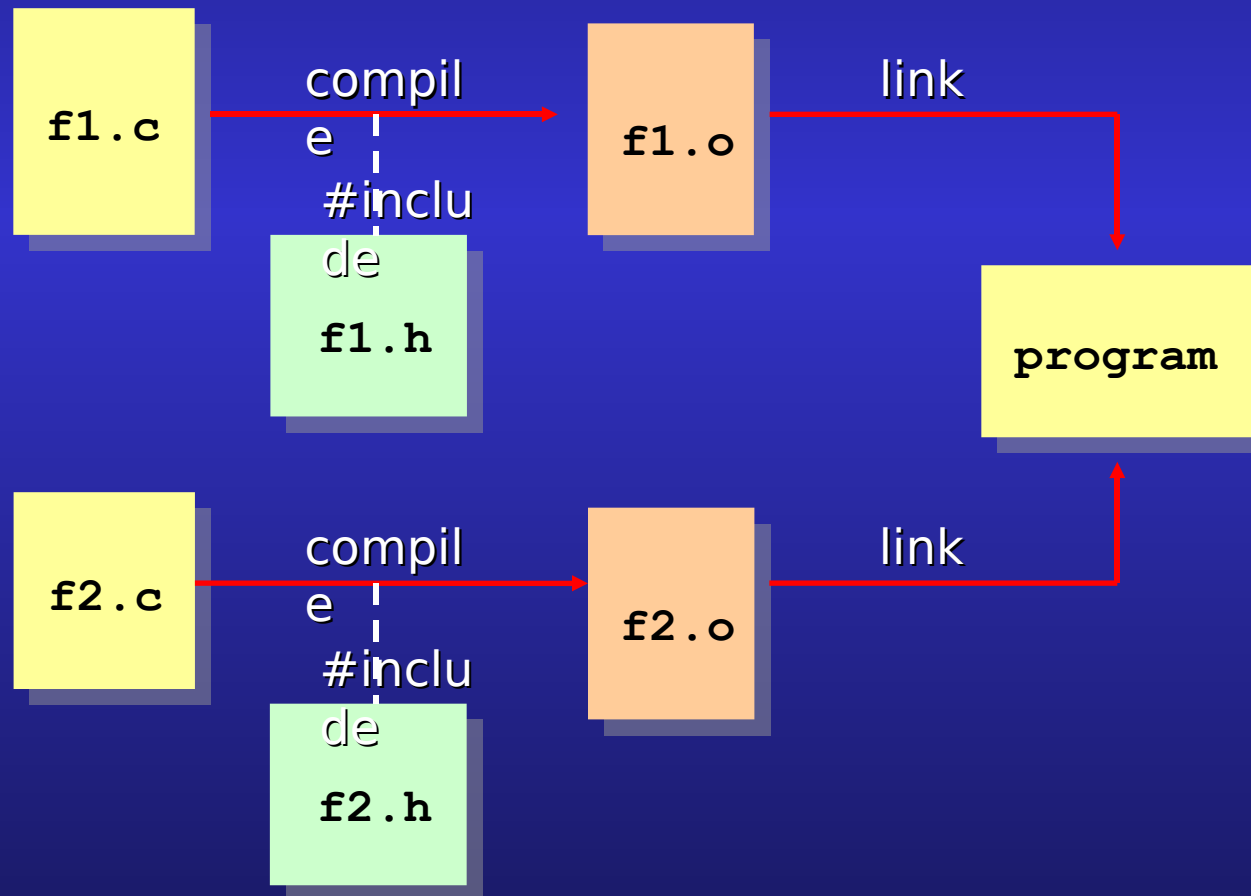
```
int i = 6;  
i = ++i;  
i = i++;
```

what are the values of i after each assignment?

```
f(g(), h());  
f(h(), g());
```

what is the difference in the order of function calls in these two statements?

- **.h header file**
 - ◆ declares the existence of specific constructs
- **.c source file**
 - ◆ defines the constructs promised in the .h file



- **a common layout style is...**
 - ◆ one space on either side of a binary operator
 - ◆ one space after a comma but not before
 - ◆ one space after each keyword
 - ◆ each statement on a separate line
 - ◆ four spaces per indent
 - ◆ physical indentation == logical indentation
 - ◆ no space before a semicolon
 - ◆ no spaces around function-call parentheses
 - ◆ no space between a unary operator and its operand
 - ◆ no tabs
- **lay out tokens to reflect grammatical structure**
 - ◆ be consistent
 - ◆ be conventional
 - ◆ english only

- **code level**
 - ◆ use comments to say things you can't say in code
 - ◆ somefile.c should always include its own header file as the very first line (`#include "somefile.h"`)
- **process level**
 - ◆ edit and compile in small cycles
 - ◆ compile with maximum warnings
 - ◆ consider treating warnings as errors
 - ◆ make sure header files compile in their own right
- **unit-testing**
 - ◆ no one aspect of software development is more encompassing than testing
 - ◆ automatic testing is compulsory

- This course was written by

Expertise: Agility, Process, OO, Patterns
Training+Designing+Consulting+Mentoring

{ JSL }

Jon Jagger

jon@jaggersoft.com

www.jaggersoft.com