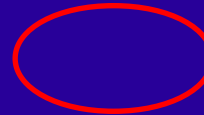
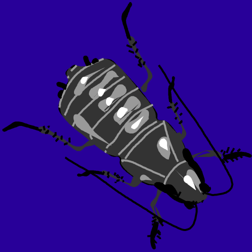
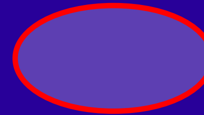
A Brief Tour

symbol key

	compiles		we're happy
	compiles but questionable		we're not happy
	doesn't compile		highlight
	a bug		key to the nature of C
	a note		introduced in 1999

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

tokens

there are 5 kinds of tokens in C programs

token	*examples*
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 // c99

mutli-line comment

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

single-line comment

identifiers

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

```
answer42
42Answer
```

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

```
different
Different
```

```
one_style
anotherstyle
YetAnother
```

```
~~strt~~
start
```

keywords

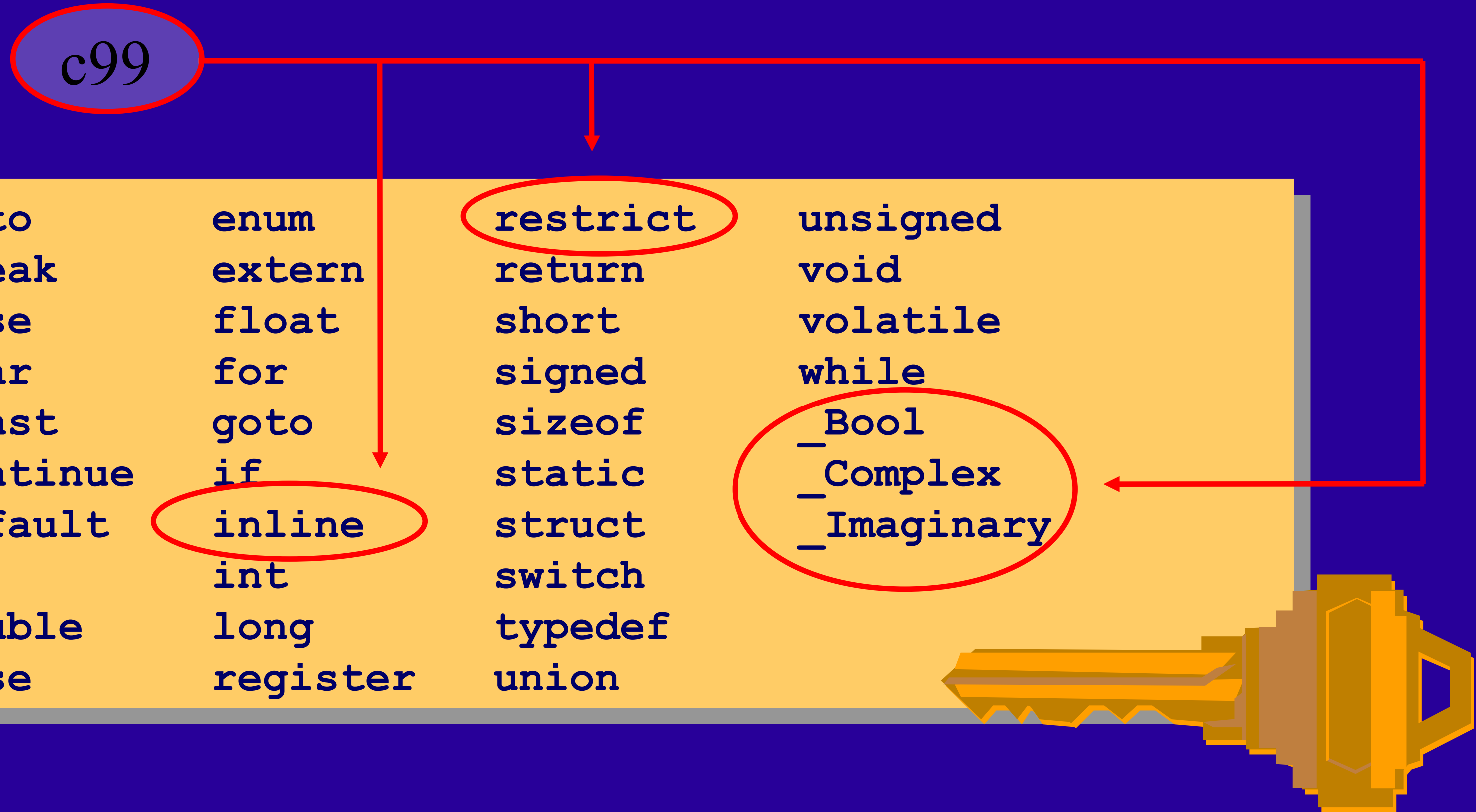
- some identifiers have a fixed meaning
  - the easy part of learning the language is learning these fixed meanings

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

int main(void)
{
    printf("Hello, world\n");
    return 0;
}
```

keyword table

- 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



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	

literals

- some tokens represent specific values

type	literal	
`bool`	`true false` c99	truth
`double`	`3.1415`	floating point
`int`	`9 6 13 42`	integers
`char`	`'X'`	characters
`char *`	`"Hello"`	sequences of characters

operators

- some tokens represent specific actions

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

separators

- 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**

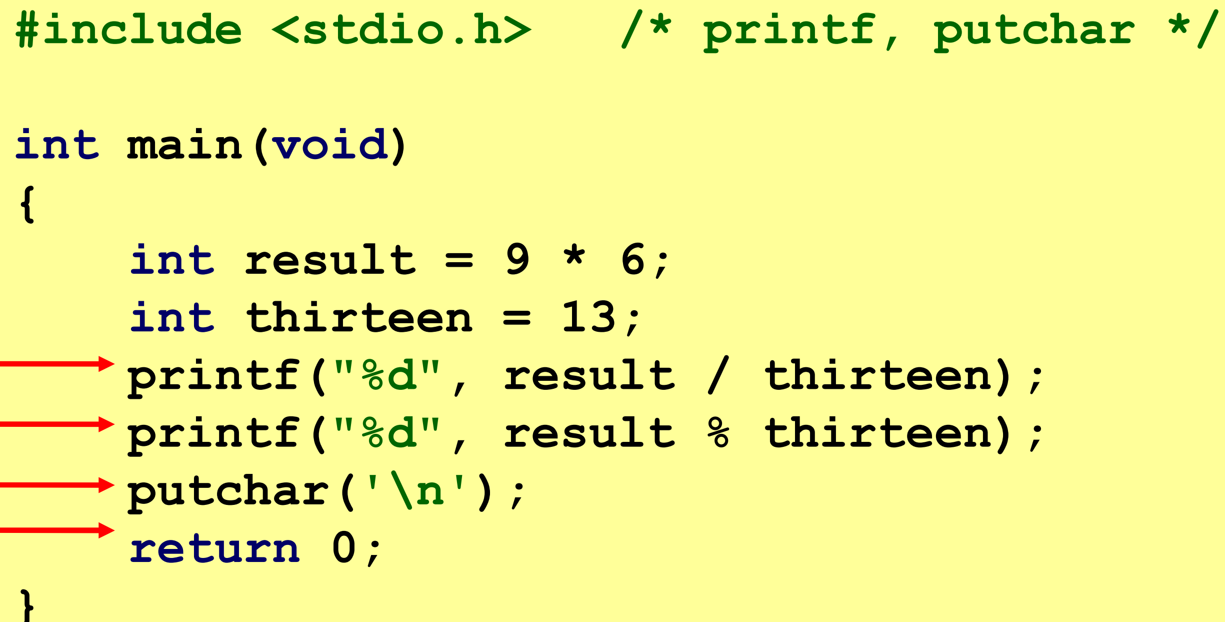
```
type identifier ;
```

```
type identifier, another ;
```

```c
#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



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

argc is the argument count

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

- 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?

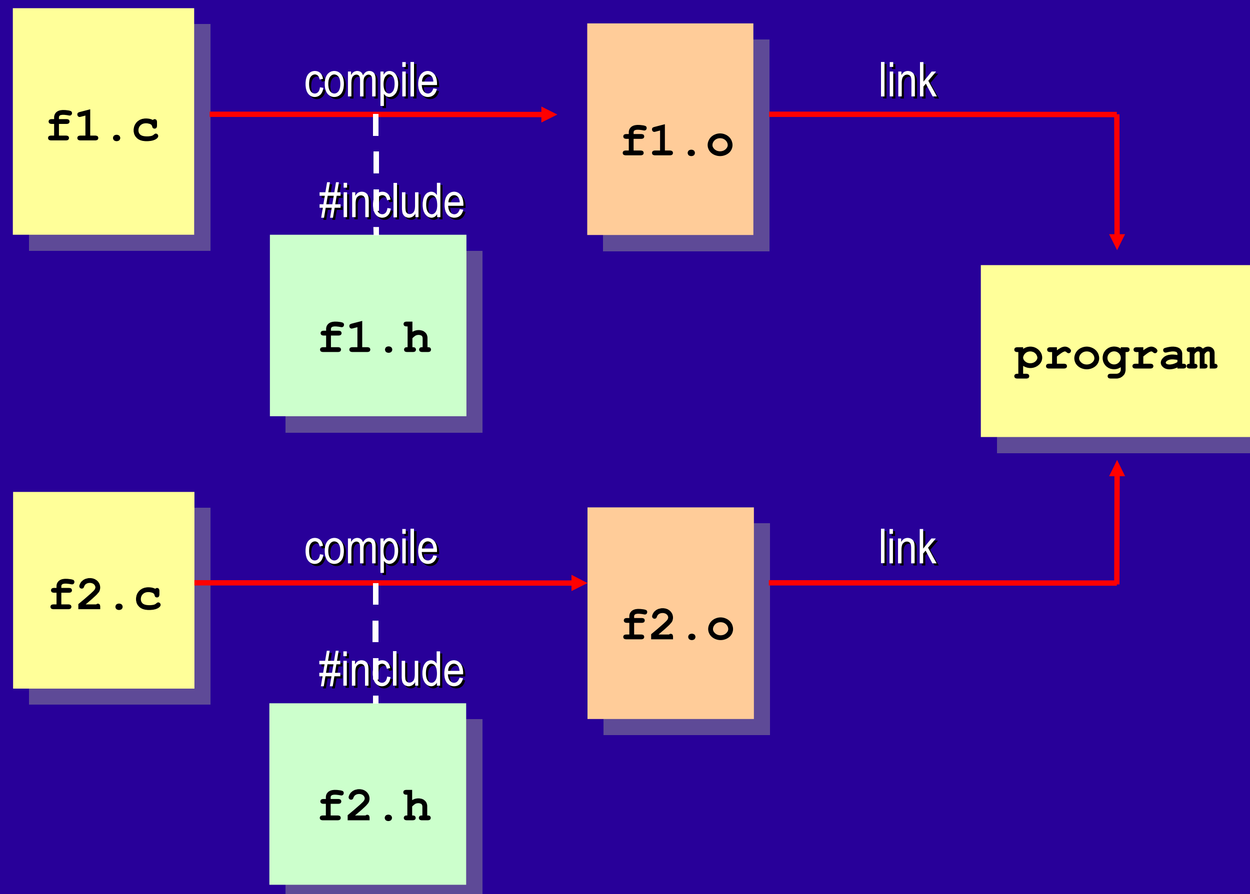
```
int i = 6;
i = ++i;
i = i++;
```

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
  - define the constructs promised in the .h file

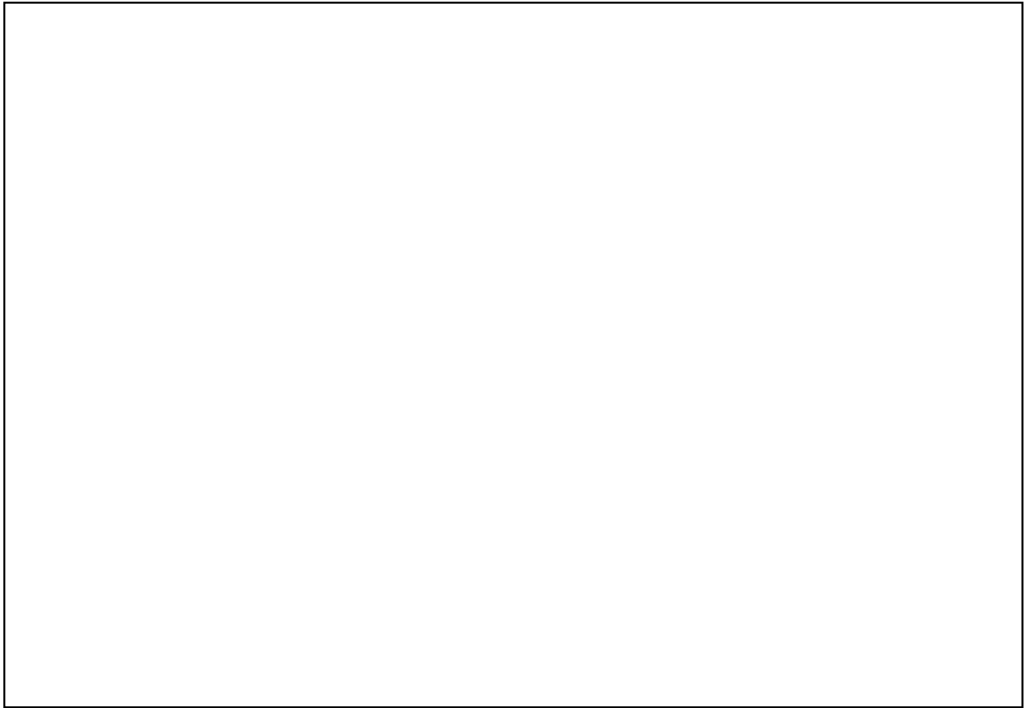


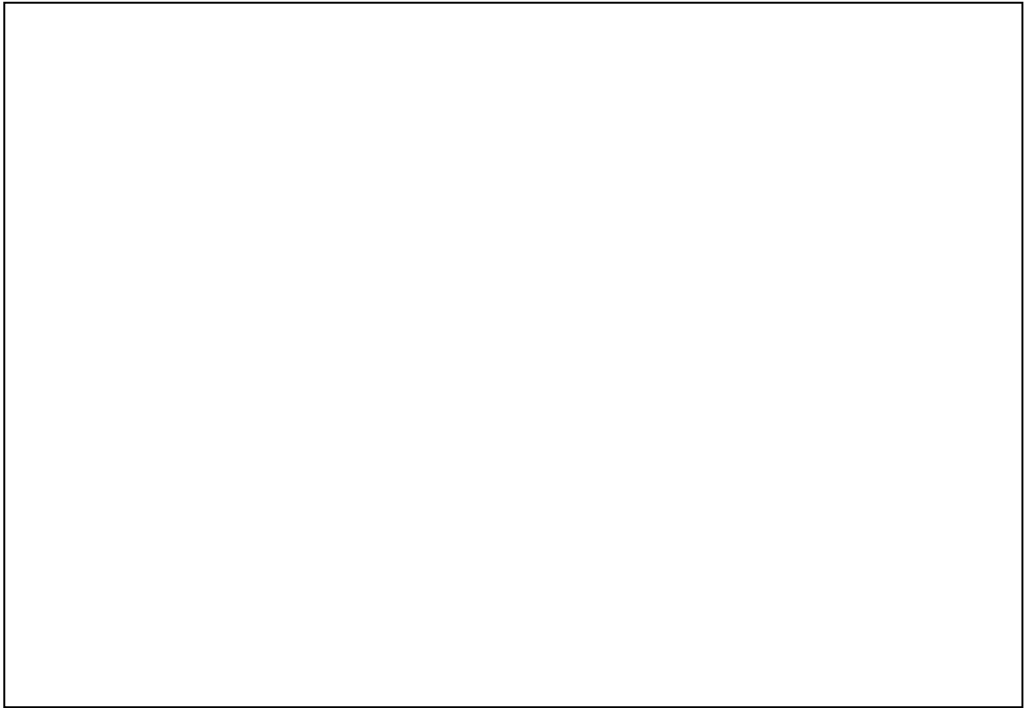
layout

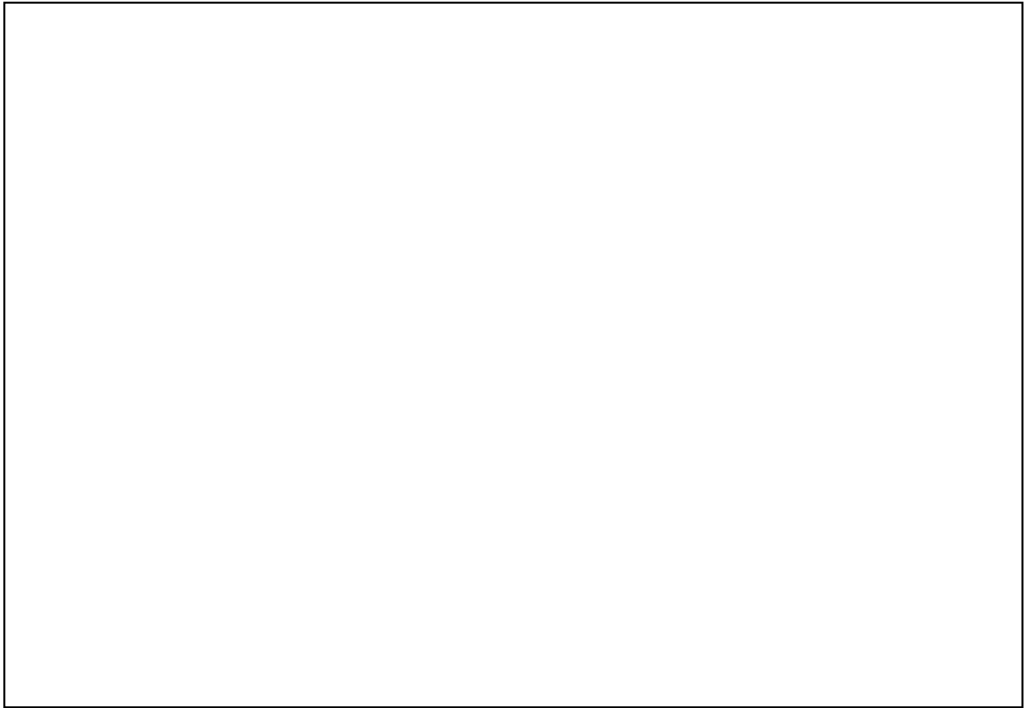
- a common layout style is…
  - one space on either side of a binary operator
  - one space after a comma but not before
  - one space after each keyword
  - each statement on a separate line
  - four spaces per indent
  - physical indentation == logical indentation
  - no space before a semicolon
  - 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

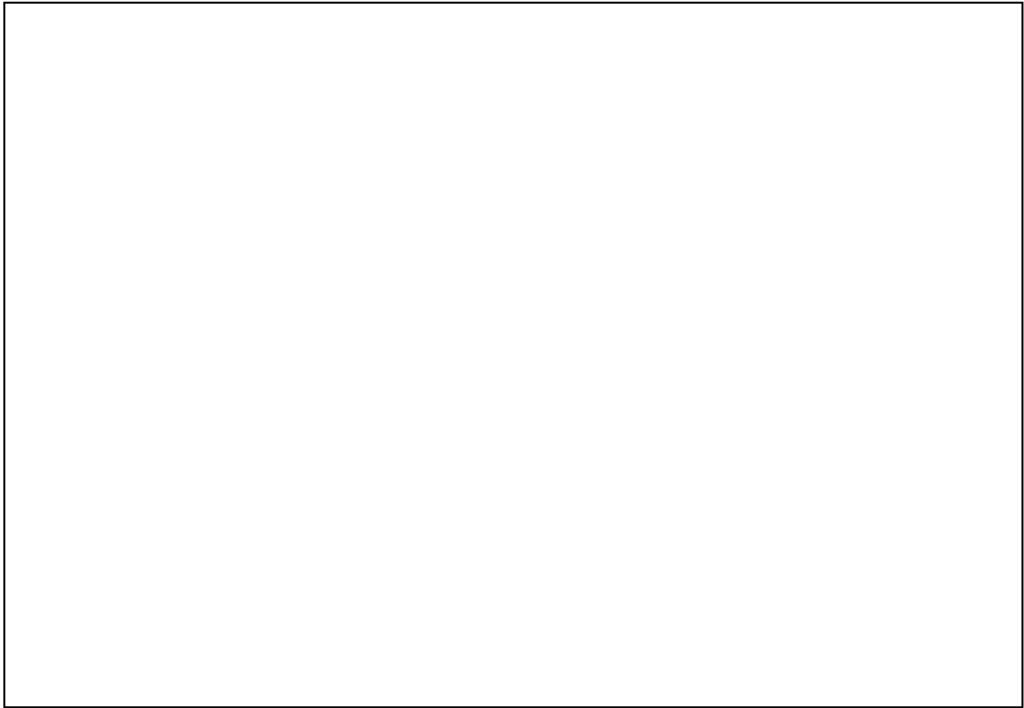
guidelines

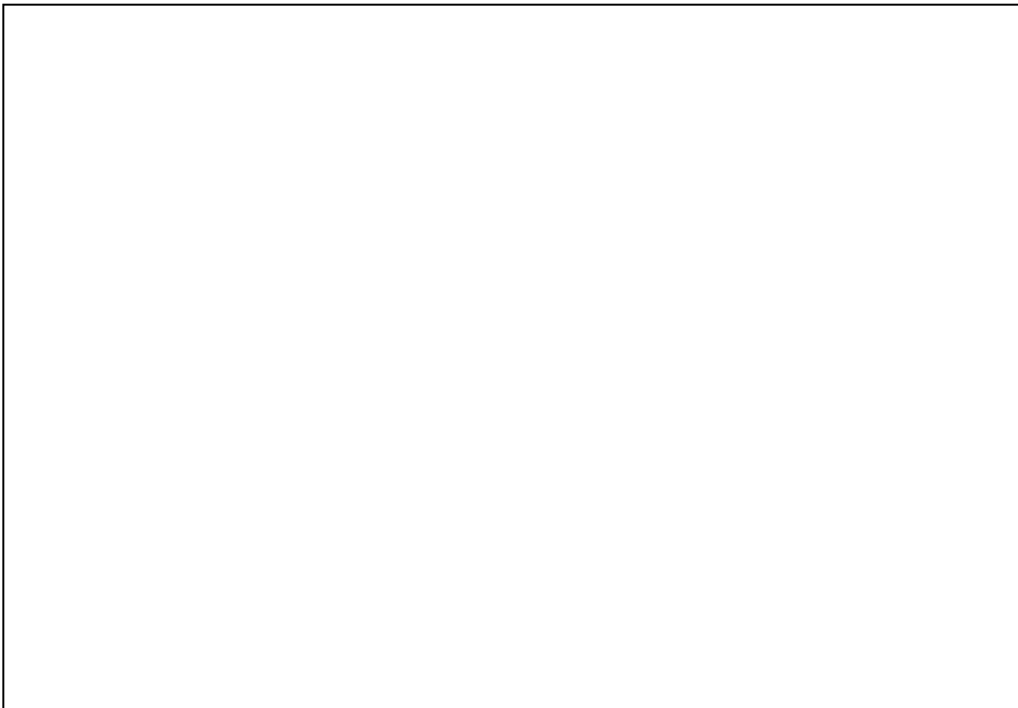
- code level
  - use comments to say things you can't say in code
  - somefile.c should always include its own header file as the very first line (#include "somefile.h")
- process level
  - edit and compile in small cycles
  - compile with maximum warnings
  - consider treating warnings as errors
  -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











Multi line comments do not nest. This means that the following will not compile:

```
/* outer comment
    int v1 = 42;
    /* inner comment */
    int v2 = 24;
*/
```

This compile will see the `*/` after inner comment as the end of the `/*` outer comment. It will then see a declaration of an `int` variable called `v2` initialised to `24`. This declaration is perfectly fine. But then the compiler will find the `*/` and that will cause it to fail.

It is possible to use the preprocessor to skip sections of code regardless of whether it contains comments or not:

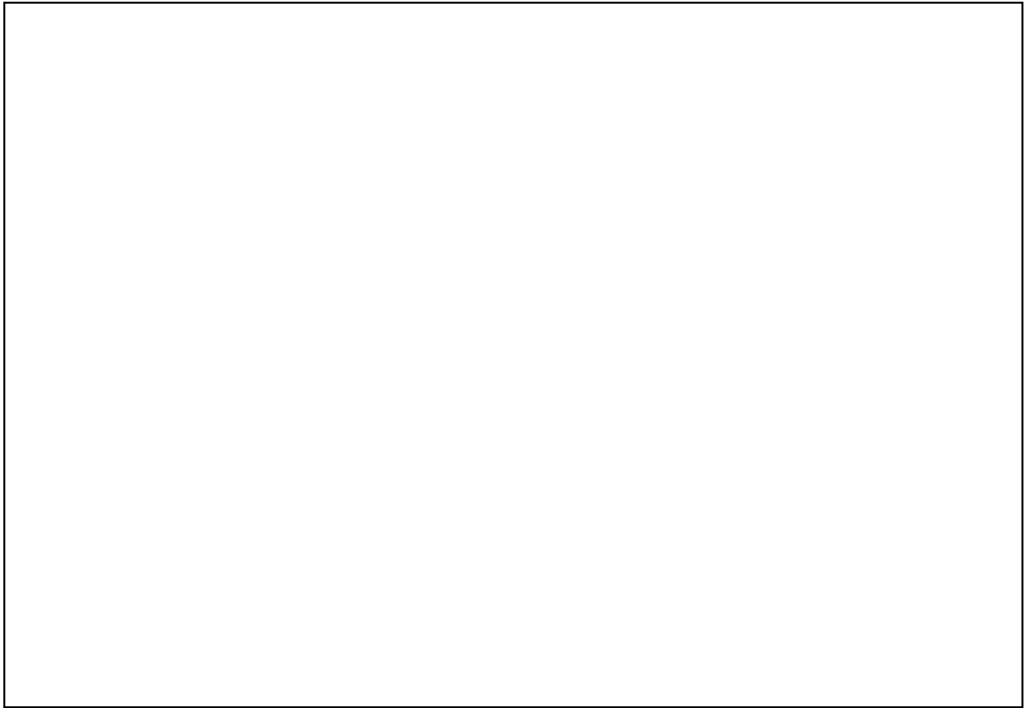
```
#if 0
 int v1 = 42;
    /* inner comment */
    int v2 = 24;
#endif
```

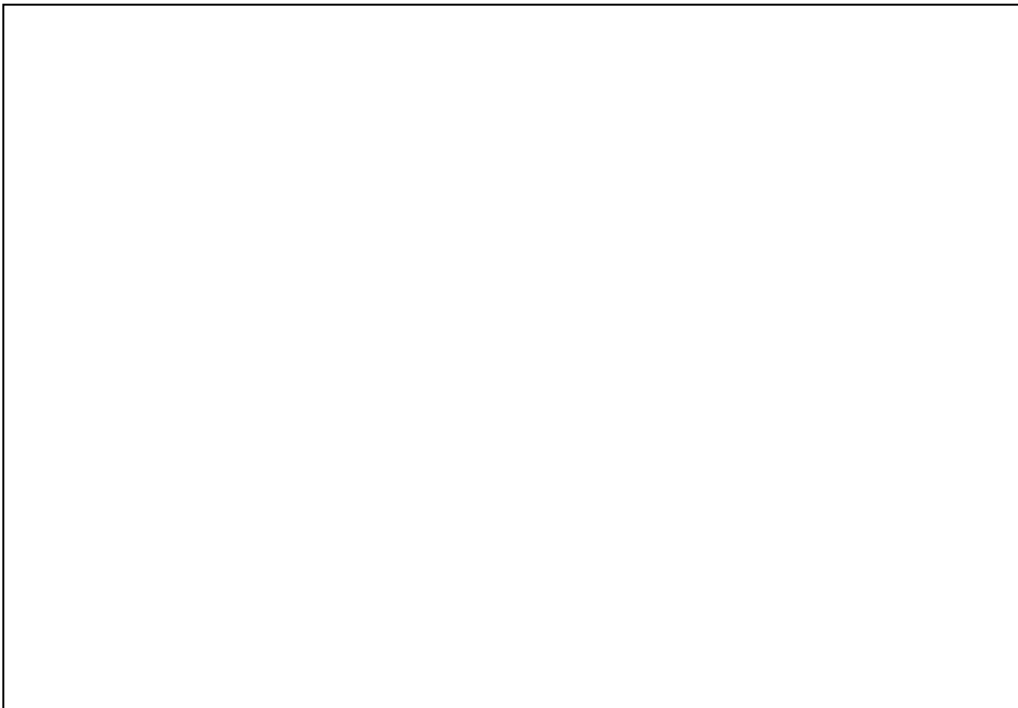
Note that the single-line comments are new with C99, although they have been supported as a common extension in many pre-C99 compilers.

The C standard library reserves identifiers starting with two underscores or a single underscore followed by an upper-case letter for itself.

For what it's worth your author prefers the style where multiword identifiers are separated by underscores. This avoids problematic identifiers such as `onexit` and `atexit`. Is that `on_exit` or `one_xit`? Is it `at_exit` or `ate_xit`? What is a `xit`? Can you eat one?

When it comes to spelling, be as conventional and consistent as possible and try to avoid abbreviations. For example, if you mean window, then say `window` (or `Window`), not `win`, `wind`, `wnd`, etc. Some abbreviations are in common use in a domain, but even here there can be scope for confusion, so think twice before using one. And don't be too clever: if you have a variable called `count` and you need another variable with a similar intent, don't call it `kount` (or even `count2`).



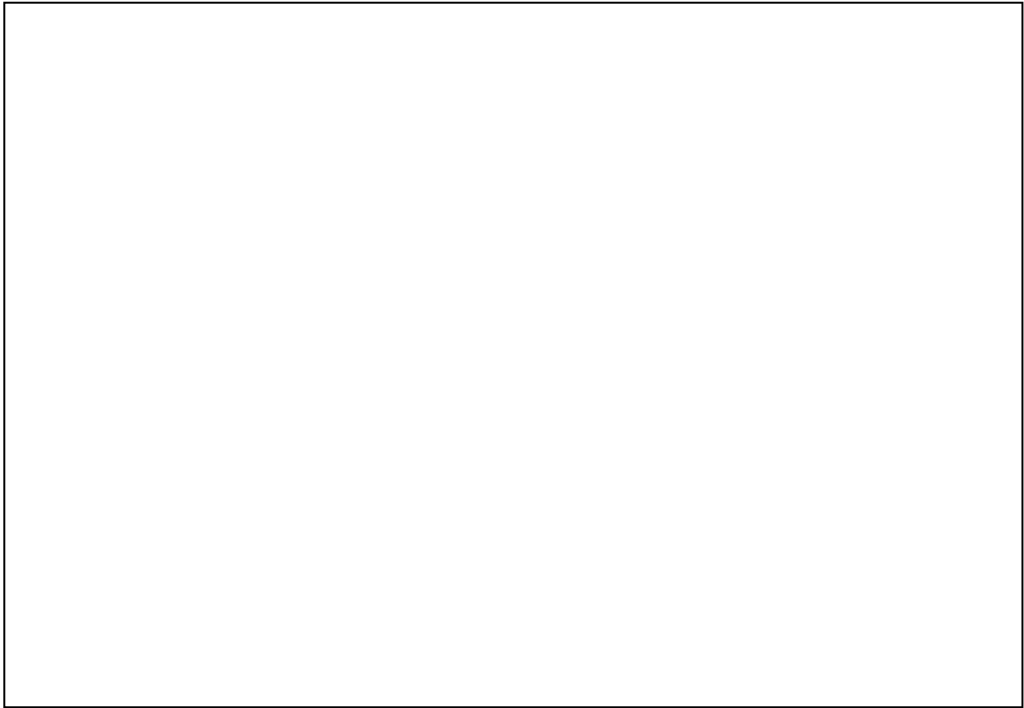


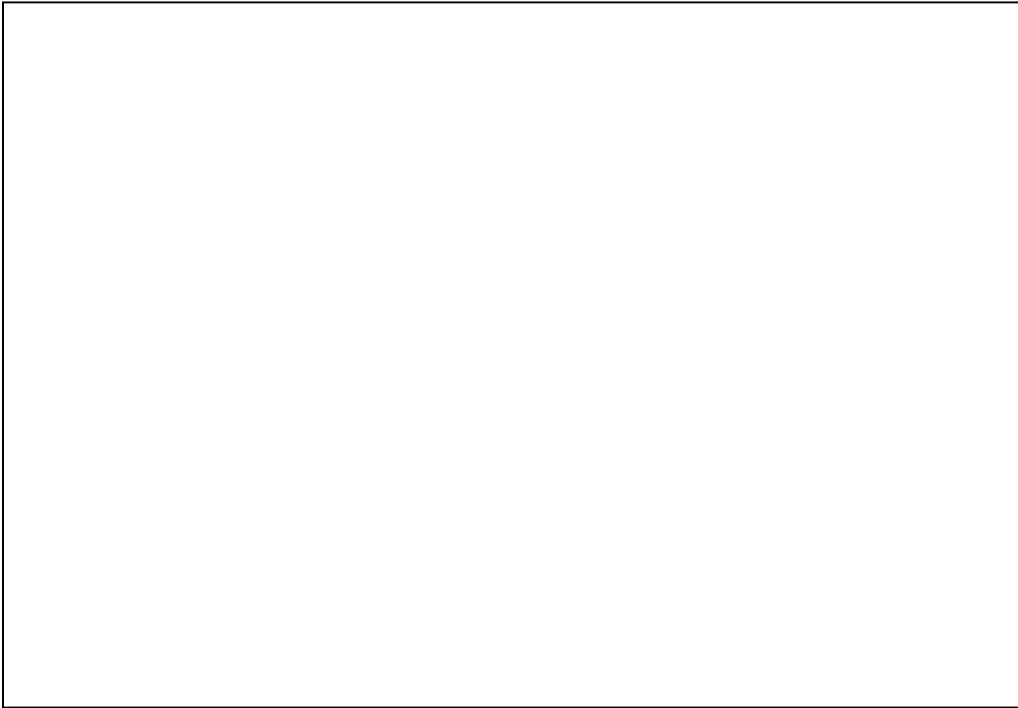
The keywords `const`, `enum`, `void` and `volatile` were added in C89.

`<stdbool.h>` was added in C99. Including it introduces the macros `bool`, `true` and `false`.

Here are some C++ keywords not found in C: `asm`, `catch`, `class`, `delete`, `explicit`, `export`, `friend`, `mutable`, `namespace`, `new`, `operator`, `private`, `protected`, `public`, `template`, `this`, `throw`, `try`, `typeid`, `typename`, `using`, `virtual`.

Beware that some keywords that are common to C and C++ have different meanings.



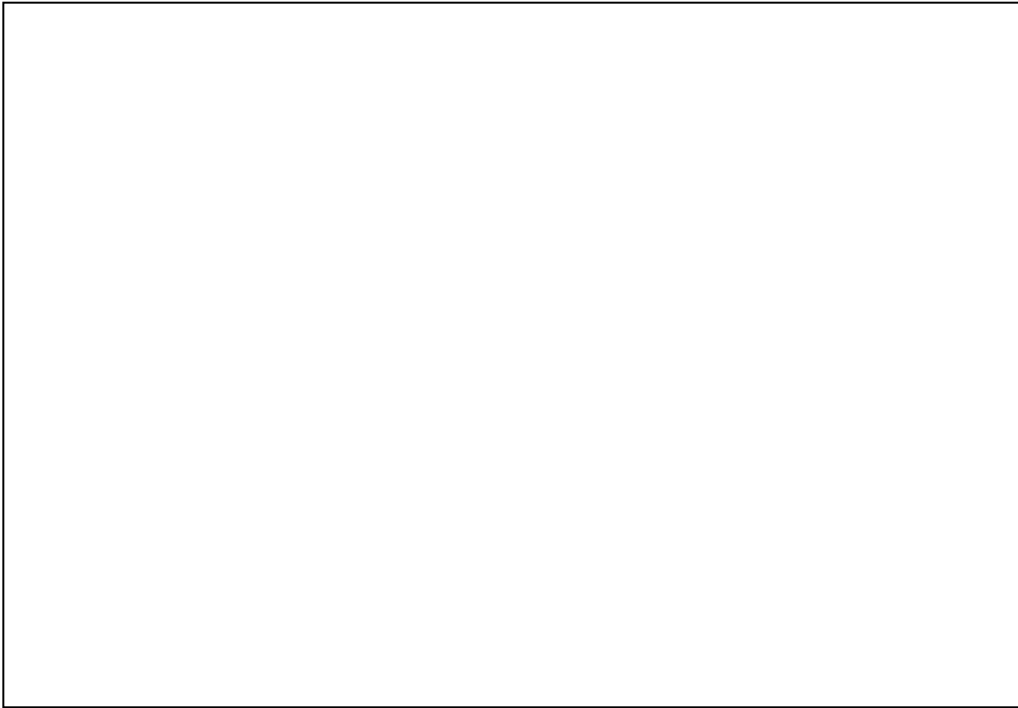


C has a rich set of operators making it a very expressive language.

The semi-colon at the end of a simple statement is how you say that you have finished the statement and that you wish to discard any remaining value that is the result of the whole statement expression. For example, the printf function returns an int value, but this value is not being used in a larger expression.

Some terminology: rounded brackets `( )` are called parentheses; curly brackets `{ }` are called braces; square brackets `[ ]` are called square brackets!

C95 introduced digraphs for those with challenged keyboards. The digraphs for `{` and `}` are `<%` and `%>` respectively.

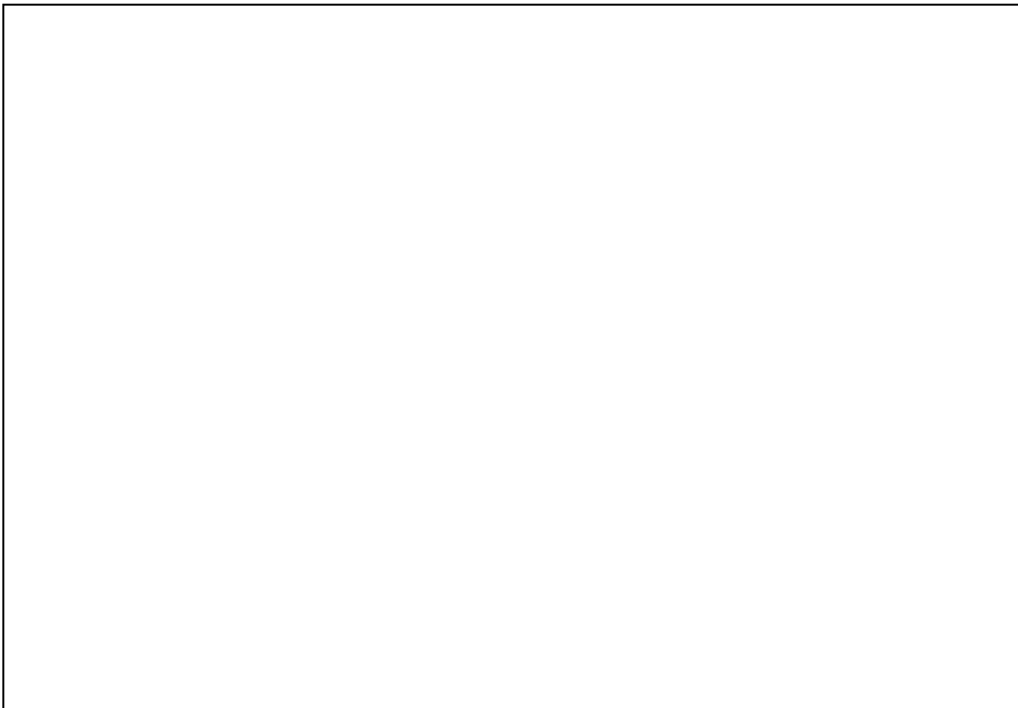


In C89 all declaration statements must precede all expression statements in a block. In C99 this rule was relaxed and the two kinds of statements can be interspersed.

An example of an expression that need not have a side-effect is a simple literal such as `42`. Thus a statement such as `42;` is a valid C statement which does absolutely nothing. If you compile with the correct settings you should get a warning saying "statement with no effect" or words to that effect.

The allowed signatures for main are:

```
int main( );
int main(void);
int main(int, char * [])
int main(int, char * *);
```



Except for operands for short-circuiting operators (`&&`, `||` and `?:`) and the comma operator, evaluation order within expressions is unspecified. Operator precedence and associativity have no effect on evaluation order. Function-call arguments to are not necessarily evaluated in left to right order – or even a consistent order. In the case of function calls, the use of the comma as a separator should not be confused with the separate concept of the comma as a sequence operator.

Not only is it important to appreciate the (non)requirements on evaluation order, but it is important to understand that updates to a value cannot happen in a well-defined way more than once between what are known as sequence points. Sequence points align with the defined evaluation orderings.

Answering the questions above, evaluation order in arithmetic expressions is unspecified and is not influenced by any precedence of the operators or grouping of parentheses. Because it is unspecified, the evaluation order need not be the same... but it may be. In the assignment and increment exercise, `i` is updated twice before the sequence point defining the end of each statement, so the result in each case is undefined. In the last example, the only guarantee is that `f` is called last in each case. Evaluation order of the arguments is unspecified and may be the same or reversed between the two statements.

For each `.c` file aim to 'export' a corresponding `.h` file that publishes the available constructs in the `.c` file. So `f1.c` and `f2.c` publish through `f1.h` and `f2.h`, respectively. Or, looked at another way, f1.c and f2.c implement the features promised in `f1.h` and `f2.h`.