C Reserved Words

An Advanced Introduction to Unix/C Programming

Dennis Ritchie

Ken Thompson

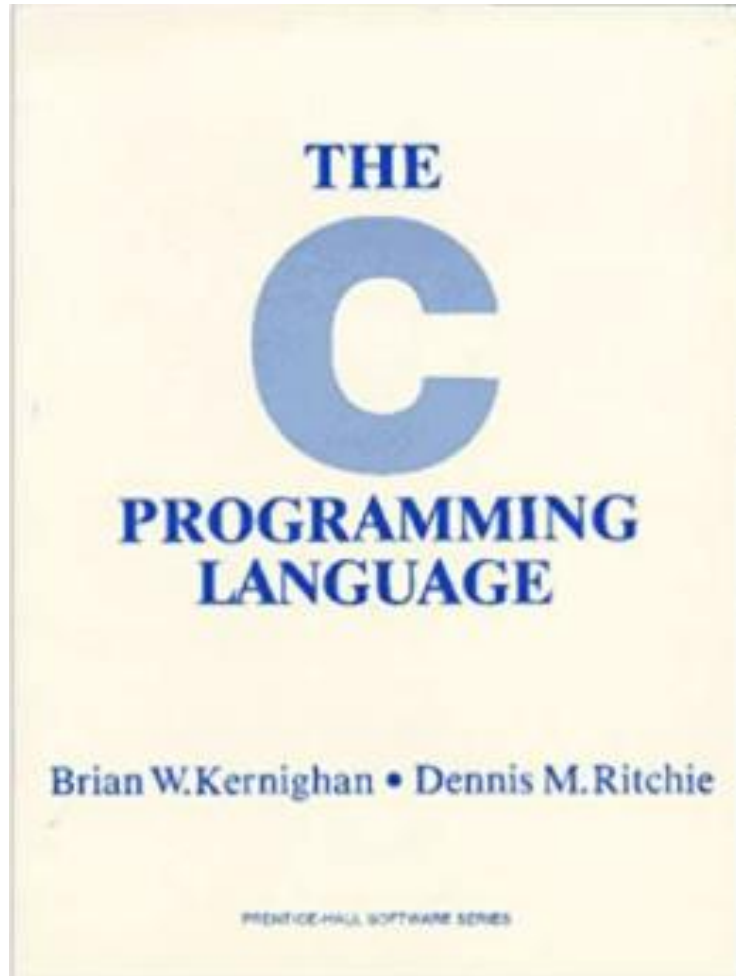
Linus Torvalds

Richard Stallman

Brian Kernighan

John Dempsey
COMP-232 Programming Languages
California State University, Channel Islands

C Programming Language Book

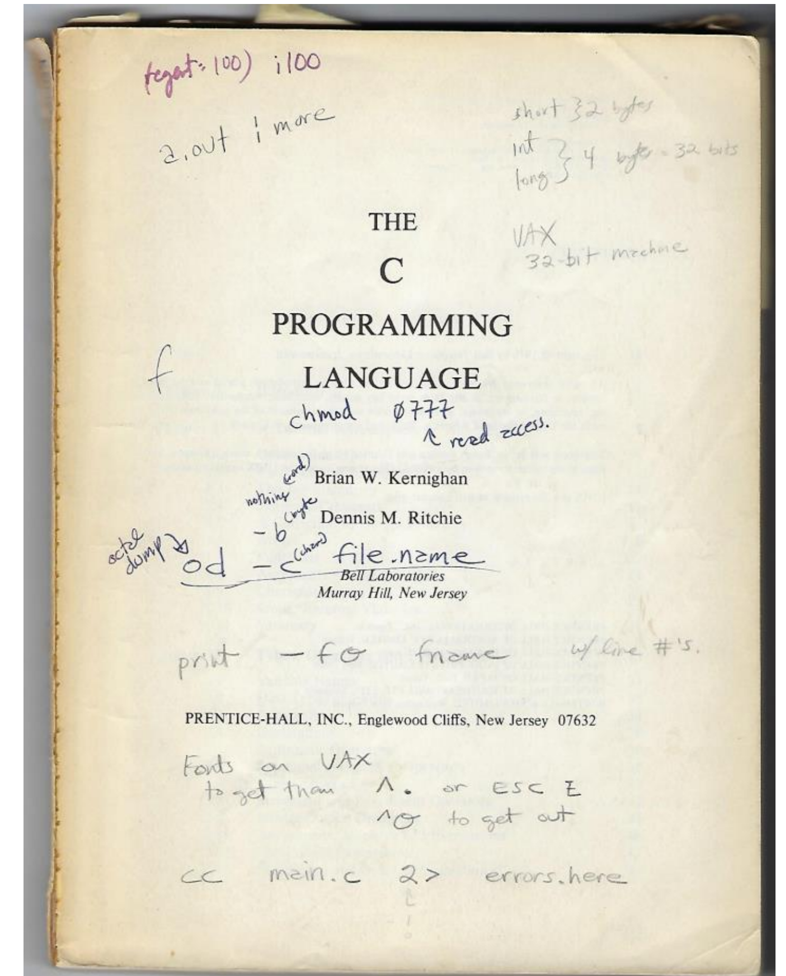


In 1978, the authors of the C Programming Language published their legendary book on C.

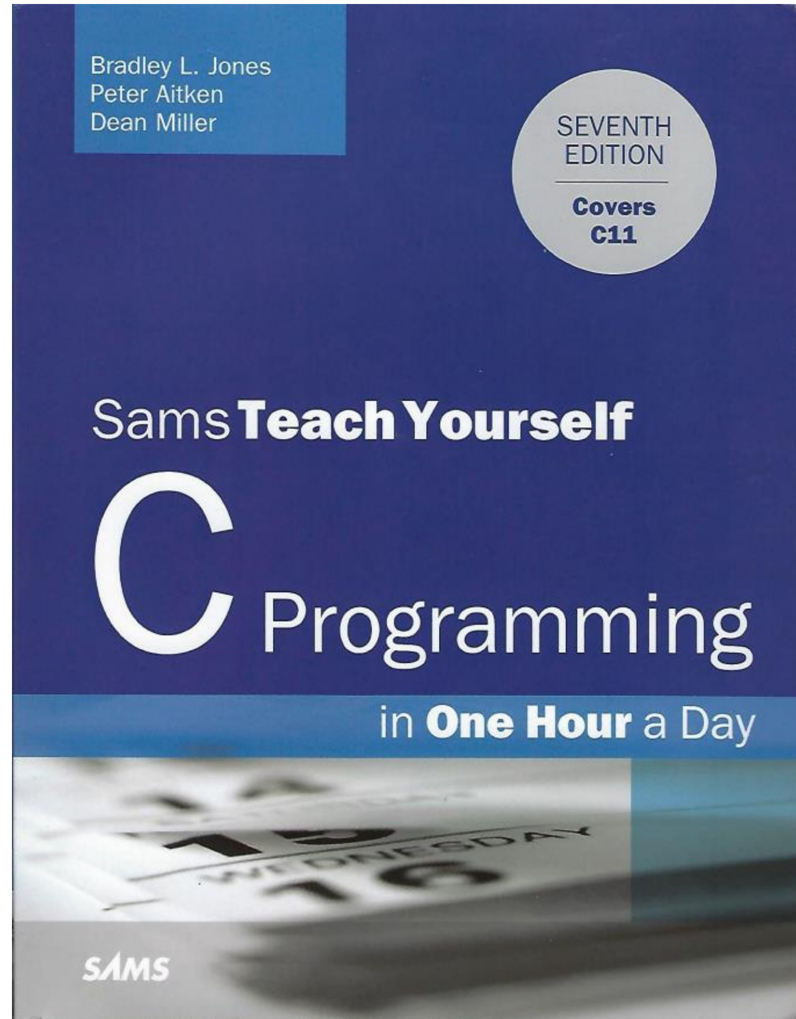
To the left is a new copy.

To the right is my copy.
(It's missing the cover.)

This is not an easy book to learn C from.

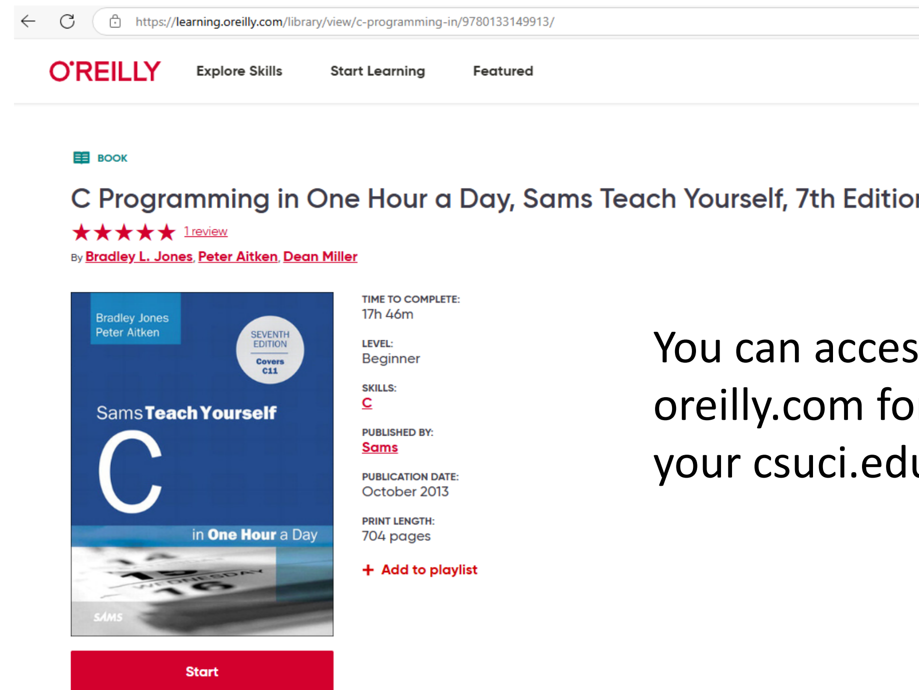


C Programming Language Book



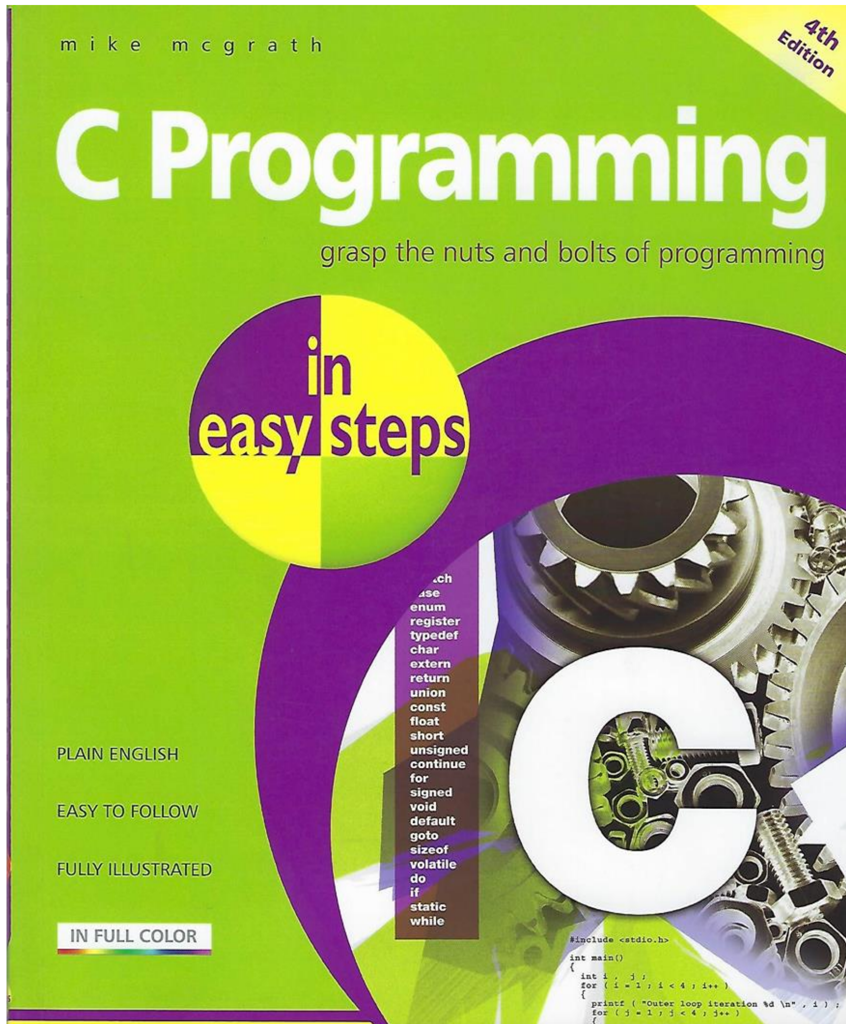
While not required, if you are looking for a good book to learn C from or use as a reference, I recommend:

C Programming in One Hour a Day



You can access this book at oreilly.com for free! Login using your csuci.edu email address.

C Programming Language Book



While not required, if you are looking for a good book to learn C from or use as a reference, I recommend:

C Programming in Easy Steps (192 pages)

Check Ebay – Sort on Price + Shipping: Lowest Price

C Features

- C was developed by Dennis Richie in 1972.
- C is a general-purpose, low level programming language, above Assembly.
- Great for system programming.
- Can access and manipulate bits, bytes, addresses, and pointers.
- Supports dividing large programs into individual modules.
- Can be integrated with modules written in different languages.
- Used for software applications that need to run fast.
- One of the most widely used programming languages today.

Worldwide 30% - July 2022

- Python is the top programming language in use today at 16.12%.
- Java is used on 8.59% of projects.
- C is the second most popular language at 9.48%.
- Languages related to C represent more than 33% of all applications.
  - C 9.48%
  - C++ 10.34%
  - C# 6.72%
  - Swift 1.13%
  - PHP 1.15%
  - Objective-C 0.94%
- 16.12% (Python) + 8.59% (Java) + 30% (C) = **55% of the marketplace**

Source: tiobe.com July 2024

Learn One, Learn All

If you can learn one programming language really, really well, then learning any other programming language will be easy to do.

In this class, the language we plan to learn really, really well is C since C is the foundation for 30% of all languages in use today.

Hello World!

- Below is the Hello World program:

```
#include <stdio.h>            ← Copies stdio.h from /usr/include/stdio.h
void main()                   ← main returns no value.
{
    printf(“Hello World!\n”);         ← \n prints a new line.
}
```

To Compile and Run

- To create and compile hello.c file, you can run:

% **vi hello.c** ← The visual editor vi

% **gcc hello.c –o hello** ← GNU C Compiler/Linker

% **hello** ← To run the hello program

Comments?

- Yes, but of course. Two ways: // and /* */

```
// This entire line is commented out.
    i = 10;          // The rest of this line is commented out.

/*
   All text between the / * and * / is commented out.
   Including this line too.
*/
```

But which comments should you use?

Comments

Use // when you can.

Using /* and */ can comment out an entire section of code, including // commented out lines. Useful for debugging purposes.

Use /* and */ when you need to write a header section or a paragraph explaining how your program works.

C Reserved Keywords

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

asm - Inline Assembly Language

```
john@oho:~$ cat a.c
#include <stdio.h>
int main() {
	int arg1, arg2, add, sub, mul, quo, rem ;

	printf( "Enter two integer numbers to add: " );
	scanf( "%d%d", &arg1, &arg2 );

	__asm__ ( "addl %%ebx, %%eax;" : "=a" (add) : "a" (arg1) , "b" (arg2) );

	printf( "%d + %d = %d\n", arg1, arg2, add );
	return 0 ;
}
john@oho:~$ gcc a.c; a.out
Enter two integer numbers to add: 10 20
10 + 20 = 30
```

auto

```#include <stdio.h> int main() { auto int number = 5; { auto int number = 20; printf("inner number: %d\n", number); } printf("outer number: %d\n", number); return 0; }```	```#include <stdio.h> int main() { int number = 5; { int number = 20; printf("inner number: %d\n", number); } printf("outer number: %d\n", number); return 0; }```

```c
#include <stdio.h>
int main()
{
      auto int number = 5;
      {
        auto int number = 20;
        printf("inner number: %d\n", number);
      }
      printf("outer number: %d\n", number);
      return 0;
}
```

```
john@oho:~$ gcc auto.c; a.out
inner number: 20
outer number: 5
```

```c
#include <stdio.h>
int main()
{
      int number = 5;
      {
        int number = 20;
        printf("inner number: %d\n", number);
      }
      printf("outer number: %d\n", number);
      return 0;
}
```

```
john@oho:~$ gcc auto2.c; a.out
inner number: 20
outer number: 5
```

auto stands for Automatic Storage Duration. This is the default when defining any local variable.
Practical usage: NONE
Why does it exist? Backward compatibility with older versions of C code which used auto.

C Numeric Data Types

Data Type	# Bytes Used	Value Range	Unsigned Value Range
char	1 byte (8 bits)	-128 to 127	0 to 255
short int	2 bytes	-32,768 to 32,767	0 to 65,535
int	4 bytes	-2,147,483,648 to 2,147,438,647	0 to 4,294,967,295
long int	4 bytes	Same as above	Same as above
long long int	8 bytes	-2^{63} to $2^{63}-1$	0 to $2^{64}-1$
float	4 bytes		
double	8 bytes		

C Numeric Data Types Abbreviated

Abbreviated	Full Name
char	signed char
short	signed short int
int	signed int
long	signed long int
long long	signed long long int
unsigned char	unsigned char
unsigned short	unsigned short int
unsigned int	unsigned int
unsigned long	unsigned long int
unsigned long long	unsigned long long int

printf/sprintf Format Specifiers

Specifier	Type
%c	character
%d	int, short
%l	long
%f	float
%s	string, character arrays
%u	unsigned int
%lu	unsigned long int
%p	pointer address
%o	octal
%x, %X	hexadecimal
%%	% character

Examples:

printf("Value = %d\n", 10);

sprintf(string, "Float value = %f\n", 3.14);

printf("ptr points to %p\n", ptr);

printf("ptr points to %x\n", ptr);

printf/sprintf Escape Sequences

\a	Rings Bell	**Examples:**
\n	New Line	printf(“Hello World!\n”);
\r	Carriage Return	
\t	Tab	sprintf(string, “\a\t\tThis is indented.\n”);
\\	Backslash	
\?	Question Mark	
\’	Single Quote	
\”	Double Quote	printf(“This is \”double quoted\” text.\n”);

Constants

- Two ways. Using #define is more common.

```
#define MAX_CONNECTIONS     16
#define PI                  3.14159;

const short int MAX_CONNECTION = 16;
const float pi = 3.14159;
```

Assignment

You can write:	Or you can write:

`int a = 5, b = 10, y, z;`
`int x = y = z = a + b;`
	`int a = 5;`
	`int b = 10;`
	`int c;`
	`int x;`
	`int y;`
	`int z;`
	
	`x = a + b;`
	`y = x;`
	`z = x;`

While both work, which do you prefer?

Assignment

This is my preferred way:

```
int     a = 0;       ← Variables are alphabetically defined.
int     b = 0;       ← Those that need to be initialized are.
int     x;
int     y;
int     z;

x = a + b;           ← You don't hide setting y and z.
y = x;               ← I find this to be much clearer to read.
z = x;               ← And as such, please follow this format.
```

Preprocessor

Preprocessor will:

1. Remove comments
2. Expand macros

Preprocessor outputs modified source code to be compiled.

#include <stdio.h> ← The stdio.h file found in /usr/include is copied into the source code prior to being compiled.

#define MAX 16 ← MAX is replaced in the source code with 16.

Preprocessor

```
#define DEBUG                    ← Defines DEBUG variable.

#ifdef DEBUG                     ← Checks if DEBUG is defined (it is), then …
  printf("125 c.c DEBUG: count =%d\n", count);   ← Prints the value of count.
#endif

#ifdef DEBUG
  printf("%d %s DEBUG: count = %d\n", __LINE__, __FILE__, count);
#endif
```

How else can we define DEBUG without hard coding a "#define DEBUG" statement in the source code?

Preprocessor – What the compiler sees

```
john@oho:~$ cat define.c
#include <stdio.h>

#define PI    3.14

// This is a great comment!
int main()
{
#ifdef DEBUG
        printf("%d %s DEBUG: The value of Pi is %0.02f\n",
                __LINE__, __FILE__, PI);
#else
        printf("Program define.c compiled on: %s.\n“
                    __DATE__);
#endif
}
```

```
john@oho:~$ gcc -E –DDEBUG define.c
... –E produces pre-processor output
... We’ll skip over the stdio.h content ...

int main()
{
 printf("%d %s DEBUG: The value of Pi is %0.02f\n",
          10, "define.c", 3.14);
}

john@oho:~$ gcc -DDEBUG define.c; a.out
10 define.c DEBUG: The value of Pi is 3.14

john@oho:~$ gcc define.c; a.out
Program define.c compiled on: Feb 1, 2023.
```

C Operator Precedence – High to Low

( ), [ ], ->, . — **Parenthesis, Brackets, Structure Pointer, Structure Member**

!, ~, ++, --, *, & — **Logical Not, Bitwise Not, Addition, Subtraction, Pointer, Address Of**

***, / , %** — **Multiplication, Division, Modulus Remainder Operator**

+, - — **Addition, Subtraction**

<<, >> — **Bit Shift Left, Bit Shift Right**

<, <=, >, >= — **Less Than, Less Than or Equal, Greater Than, Greater Than or Equal**

==, != — **Equal, Not Equal**

& — **Bit And**

^ — **Bit Exclusive Or (XOR)**

| — **Bit Inclusive Or**

&& — **Logical And**

|| — **Logical Or**

? : — **If Then Else**

=, +=, -=, *=, /=, %=, %=, &=, ^=, |=, <<=, >>= — **Equal, Addition, Subtraction, …**

, — **Expression Separator**

0101 ^ 0011 = 0110 = 6

Assignment Operators

x += 1; → x = x + 1; I prefer the longer version, but use both.

x /= b; → x = x / b;

john@oho:~$ **a.out**

a = x ? y : z; ← Statement being evaluated. If x=0 then a=y else a=z.

a = 0 ? 3 : 4; ← Values being evaluated. a = 4

a = 1 ? 3 : 4; ← a = 3

z = (x>y)?x:y;

z = (1>3)?1:3; ← z = 3

if (then) else

```
if (a == b)
     c = 100;

if (a <= b) {
    printf("a is less than or equal to b.\n");
}
else {
    printf("a is greater than b.\n");
}
```

if (then) else

What is printed below?

```
int a = 5;
int b = 10;

if (a = b)
    printf(“HERE: a = %d, b = %d\n”, a, b);
else
    printf(“THERE: a = %d, b = %d\n”, a, b);
```

else if …

```
char    a;
if (a == ‘A’)
    printf(“A stands for apples.\n”);
else if (a == ‘B’)
    printf(“B stands for beans.\n”);
else if (a == ‘C’)
    printf (“C stands for corn.\n”);        // Why isn’t corn spelled corns? ;-)
else
    printf(“a isn’t A, B, or C.\n”);
```

switch/case Statement

```
switch( type ) {
        case 'a':
        case 'A':  printf("A is for apples.\n");
                   break;
        case 'b':
        case 'B':  printf("B is for beans.\n");
                   break;
        case 'C': printf("C is for corn.\n");
                   break;
        default:  printf("type isn't A, B, or C.\n");
                   break;
}
```

Does the default have to be at the bottom?

for Loops

```
for (i=0; i<20; i++)
    printf("i = %d. This is fun.\n", i);

for (a=10; a>5; a--)
    for (b=0; b<5; b++)
        printf("a = %d, b = %d\n", a, b);

printf("Final number = %d %d %d\n", i, a, b);
```

← What numbers are printed?

20 5 5

while Loop

```
#include <stdio.h>
#define MAX_COUNT      10

void main()
{
        int count = 1;

        while (count < MAX_COUNT) {
            printf(“count = %d\n”, count);
            count = count + 1;
        }
}
```

for Loop

```
for (i=0; i<20; i++)
    printf("i = %d. This is fun.\n", i);
```

The above for loop can be represented as a while loop, like so:

```
i = 0;
while (i<20) {
    printf("i = %d. This is fun.\n", i);
    i++;
}
```

do … while Loop

```
#include <stdio.h>

void main()
{
        int     i = 0;

        do {
            printf("i = %d\n", i);
            i = i + 1;
        } while (i < 10);
}
```

break

```
john@oho:~$ cat break.c
#include <stdio.h>
#include <string.h>
#define MAX_NAMES  8
char first_name[MAX_NAMES][10] =  { "Able", "Howard", "Sammy", "Lucy", "Charlie", "John", "Tom", "Sally" };
int main() {
        int i;
        for (i=0; i<MAX_NAMES; i++) {
            if (strcmp("John", first_name[i]) == 0) {
                  printf("John is record %d.\n", i+1);
                  break;                                            ← Break out of the for loop when John is found.
           }
     }
}
john@oho:~$ gcc break.c; a.out
John is record 6.
```

continue

```
john@oho:~$ cat continue.c
#include <stdio.h>
int main () {
	int a = 0;
	printf("Numbers evenly divisible by 5 up to 50:");
	do {
	  if ((a % 5) != 0) {
	    a = a + 1;
	    continue;			← If remainder not equal to zero, continue.
	  }
	  printf(" %d", a++);
	} while (a <= 50);
	printf(".\n");
}

john@oho:~$ gcc continue.c; a.out
Numbers evenly divisible by 5 up to 50: 0 5 10 15 20 25 30 35 40 45 50.
```

extern

```
john@oho:~$ cat e1.c
#include <stdio.h>

void    set_number_of_clients();

int     number_of_clients = 1000;

void main()
{
   printf("Number of clients = %d\n", number_of_clients);
   set_number_of_clients();
   printf("Number of clients = %d\n", number_of_clients);
}
```

```
john@oho:~$ cat e2.c
int     number_of_clients = 500;

void set_number_of_clients()
{
   number_of_clients = 250;
}
```

john@oho:~$ **gcc e1.c e2.c**

/usr/bin/ld: /tmp/cc6hQRmE.o:(.data+0x0): **multiple definition of `number_of_clients';**
/tmp/ccezinhF.o:(.data+0x0): first defined here

collect2: error: ld returned 1 exit status

extern

```
john@oho:~$ cat e1.c
#include <stdio.h>

void    set_number_of_clients();

int     number_of_clients = 1000;

void main()
{
   printf("Number of clients = %d\n", number_of_clients);
   set_number_of_clients();
   printf("Number of clients = %d\n", number_of_clients);
}
```

```
john@oho:~$ cat e2.c
extern int      number_of_clients;

void set_number_of_clients()
{
   number_of_clients = 250;
}
```

```
john@oho:~$ gcc e1.c e2.c; a.out
Number of clients = 1000
Number of clients = 250
```

extern

```
john@oho:~$ cat e1.c
#include <stdio.h>

void    set_number_of_clients();

int     number_of_clients;

void main()
{
   printf("Number of clients = %d\n", number_of_clients);
   set_number_of_clients();
   printf("Number of clients = %d\n", number_of_clients);
}
```

```
john@oho:~$ cat e2.c
int     number_of_clients = 500;

void set_number_of_clients()
{
   number_of_clients = 250;
}
```

john@oho:~$ **gcc e1.c e2.c; a.out**

Number of clients = ???

Number of clients = ???

extern

```
john@oho:~$ cat e1.c
#include <stdio.h>

void    set_number_of_clients();

int     number_of_clients = 1000;

void main()
{
   printf("Number of clients = %d\n", number_of_clients);
   set_number_of_clients();
   printf("Number of clients = %d\n", number_of_clients);
}
```

```
john@oho:~$ cat e2.c
int      number_of_clients = 500;

int calculate_number_of_clients()
{
   number_of_clients = 250;
   return(number_of_clients);
}
```

john@oho:~$ **gcc e1.c e2.c; a.out**

Number of clients = ???

Number of clients = ???

Previous Slide:

john@oho:~$ gcc e1.c e2.c; a.out
Number of clients = 500
Number of clients = 250

Previous Slide:
john@oho:~$ **gcc e1.c e2.c**
/usr/bin/ld: /tmp/cc1DHtRx.o:(.data+0x0): multiple definition of `number_of_clients';
/tmp/cc2cKyzw.o:(.data+0x0): first defined here
collect2: error: ld returned 1 exit status

goto

```
john@oho:~$ cat goto.c
#include <stdio.h>

void main()
{
	int	i;

	for (i=0; i<10; i++) {
	    for (j=0; j<20; j++) {
	        if ((i == 5) && (j == 6))
	            goto label;
	        printf("i = %d\n", i);
	    }
	}
label:
	printf("Time to go to label!\n");
}
```

```
john@oho:~$ gcc goto.c; a.out
i = 0
i = 1
i = 2
i = 3
i = 4
Time to go to label!
```

In general, avoid using goto statements.

But if you must, have the label below the goto statement to avoid “spaghetti code”.

inline – What is the difference?

```
john@oho:~$ cat inline.c
#include <stdio.h>

static inline int foo(int x, int y) { return x+y; }

void main()
{
        int a = 10;
        int b = 20;

        a = foo(a, b);
        printf( "a = %d\n", a ) ;
        a = foo(a, b);
        printf( "a = %d\n", a ) ;
        a = foo(a, b);
        printf( "a = %d\n", a ) ;
}

john@oho:~$ gcc inline.c; a.out
a = 30
a = 50
a = 70
```

```
john@oho:~$ cat inline2.c
#include <stdio.h>

int foo(int x, int y) { return x+y; }

void main()
{
        int a = 10;
        int b = 20;

        a = foo(a, b);
        printf( "a = %d\n", a ) ;
        a = foo(a, b);
        printf( "a = %d\n", a ) ;
        a = foo(a, b);
        printf( "a = %d\n", a ) ;
}

john@oho:~$ gcc inline2.c; a.out
a = 30
a = 50
a = 70
```

for loop

How can I make the following for loop go 40+% faster?

```
john@oho:~$ cat r.c
#include <stdio.h>
#define MAX 1000000

void main()
{
    int i;
    int sum = 0;

    for (i=0; i<=MAX; i++)
        sum = sum + i;
    printf("The sum of 0 to %d is %d.\n", MAX, sum);
}
```

register

```
john@oho:~$ cat r.c
#include <stdio.h>
#define MAX 1000000
void main()
{
	int i;
	int sum = 0;

	for (i=0; i<=MAX; i++)
	  sum = sum + i;
	printf("The sum of 0 to %d is %d.\n", MAX, sum);
}

john@oho:~$ gcc r.c; time a.out
The sum of 0 to 1000000 is 1784293664.

real	0m0.019s
user	0m0.000s
sys	0m0.000s
```

```
john@oho:~$ cat register.c
#include <stdio.h>
#define MAX 1000000
void main()
{
	register int i;
	register int sum = 0;

	for (i=0; i<=MAX; i++)
	  sum = sum + i;
	printf("The sum of 0 to %d is %d.\n", MAX, sum);
}

john@oho:~$ gcc register.c; time a.out
The sum of 0 to 1000000 is 1784293664.

real	0m0.007s
user	0m0.000s
sys	0m0.000s
```

*** real times will vary.**

restrict

```
john@oho:~$ cat restrict.c
#include <stdio.h>
int sum(int *a, int * restrict b, int * restrict c)
{
	*b = 100;
	*c = 200;
	*a = *b + *c;
}
void main()
{
	int a = 10;
	int b = 20;
	int *c;

	c = &b;
	sum(&a, &b, c);
	printf("a = %d, b = %d, c = %d\n", a, b, *c);
}
john@oho:~$ gcc restrict.c; a.out
a = 400, b = 200, c = 200
```

← Restrict is used for code optimization.
← Restrict says the object pointed to by b is only pointed to by b.
← Restrict says the object pointed to by c is only pointed to by c.

When we use restrict with a pointer ptr, it tells the compiler that ptr is the only way to access the object pointed by it, in other words, there's no other pointer pointing to the same object.

← Expected error.
← Since both b and c point to the same object.

return

```
john@oho:~$ cat return.c
#include <stdio.h>

int sum_values(int a, int b, int c)
{
  return(a + b + c);                          ← Returns the sum of a + b + c.
}

int main()
{
   int sum;
   sum = sum_values(1, 2, 3);                 ← Function sum_values returns 6.
   printf("The sum of 1 + 2 + 3 is %d.\n", sum);
   return(100);                               ← The main program returns 100.
}
john@oho:~$ gcc return.c; a.out
The sum of 1 + 2 + 3 is 6.
john@oho:~$ echo $?
100                                           ← Here's the 100 returned from main.
```

typedef / enum / union / struct / sizeof

```
john@oho:~$ cat struct.c
#include <stdio.h>
typedef enum
{
	IDENTIFIER,
	INT,
	FLOAT,
	SEMICOLON
} TOKEN_TYPE;

typedef union
{
	long	integer;
	double	floating_point;
	char	*string;
	char	operation;
} TOKEN_VALUE;

typedef struct token
{
	TOKEN_TYPE	type;
	TOKEN_VALUE	val;
} TOKEN;

TOKEN my_token;
```

```
void main()
{
	my_token.type = INT;
	my_token.val.integer  = 123;

	printf("Size of TOKEN_TYPE    = %ld\n", sizeof(TOKEN_TYPE));
	printf("Size of TOKEN_VALUE = %ld\n", sizeof(TOKEN_VALUE));
	printf("Size of TOKEN             = %ld\n", sizeof(TOKEN));
	printf("Size of my_token         = %ld\n", sizeof(my_token));
}

john@oho:~$ gcc struct.c; a.out
Size of TOKEN_TYPE     = 4
Size of TOKEN_VALUE  = 8
Size of TOKEN              = 16
Size of my_token         = 16
```

← Why isn't this 12 bytes instead?

static

With static	Without static

john@oho:~$ **cat static.c**
#include <stdio.h>
 	
int my_func()
{
    static int counter = 0;
 	
    counter = counter + 1;
    return(counter);
}
 	
void main()
{
    printf("counter = %d\n", my_func());
    printf("counter = %d\n", my_func());
}
john@oho:~$ **gcc static.c; a.out**
counter = 1
counter = 2
	john@oho:~$ **cat static2.c**
	#include <stdio.h>
	
	int my_func()
	{
	    int counter = 0; // static not used.
	
	    counter = counter + 1;
	    return(counter);
	}
	
	void main()
	{
	    printf("counter = %d\n", my_func());
	    printf("counter = %d\n", my_func());
	}
	john@oho:~$ **gcc static2.c; a.out**
	counter = 1
	counter = 1

static – Two files: static31.c & static32.c

With static function

```
john@oho:~$ more static31.c
#include <stdio.h>
void staticFunction();
void main()
{
	staticFunction();
	printf("Done.\n");
}

john@oho:~$ more static32.c
#include <stdio.h>
static void staticFunction()
{
	printf("staticFunction called.\n");
}

john@oho:~$ gcc static31.c static32.c; a.out
/usr/bin/ld: /tmp/ccDJWs6h.o: in function `main':
static31.c:(.text+0xe): undefined reference to `staticFunction'
collect2: error: ld returned 1 exit status
a.out: command not found
```

Without static function

```
john@oho:~$ more static31.c
#include <stdio.h>
void staticFunction();
void main()
{
	staticFunction();
	printf("Done.\n");
}

john@oho:~$ more static32.c
#include <stdio.h>
void staticFunction()
{
	printf("staticFunction called.\n");
}

john@oho:~$ gcc static31.c static32.c; a.out
staticFunction called.
Done.
```

static

One of the more interesting keywords.

1. Static variables are initialized to 0 unless explicitly set to an initial numeric value.
2. Static values are saved in memory for the entire duration while the program is running.
3. Static variables are allocated memory in the global data segment, not the temporary stack segment which is used for function calls.
4. Static variables save their values even when out of scope.
5. Static global variables in a file are not accessible outside the file.
6. Static functions can only be called within the code in the file.

void

```
#include <stdio.h>

void functionA(void)           ← Function does not return a value.
{                                Function does not have any parameters.
    printf("Function A called.\n");   A warning is issued if parameters are passed in.
}
void main(void)                ← Main does not return a value.
{                                Main has no arguments when called.
        functionA();
}
```

volatile

volatile is used when you want the compiler to not optimize your code.

What You Wrote:	Compiler Optimized Code:

unsigned int status = 0;
 	
while (status == 0) {
    // Do something but status not set.
}
	unsigned int status = 0;
	
	while (1) {
	    // Do something.
	}

The compiler (seeing that status is not changed in the source code) optimizes and changes while(status==0) to be a while(1) to allow the code to run faster.

Volatile is used to tell the compiler not to optimize this variable and to force the program to check the status value each time. Volatile variables can be updated externally, e.g., being mapped to actual hardware such as an I/O port.

volatile

Volatile defined variables can change by:

1. **Global variables modified by an interrupt service routine (ISR).**
2. **Global variables modified by a multi-threaded application running multiple tasks at the same time.**
3. **Changes to memory mapped registers.**

If a hardware or software interrupt occurs while running your main program, the ISR is executed. After running the ISR, control is returned back to the main program.

```
typedef struct {
        uint32_t data0 : 12;
        uint32_t data1 : 1;
        ...
}  data_port;
#define DATA_PORT (*(volatile data_port *) 0xc006380)
DATA_PORT.data0 = 1;      // Set data0 bit DATA_PORT to 1.
```

group	base addr	offset	port name
Group 1	0xc006380	0	data
		1	enable1
		2	timeIndex
		3	updateflag

data port		
data0	bits	0:11
data1	bits	12:12
data2	bits	13:13
data3	bits	14:15
data4	bits	16:23
data5	bits	0:31