

C Programming in Linux

WANG Xiaolin

`wx672ster+c@gmail.com`

September 10, 2019

Reference Books



STEVENS W R, RAGO S A. *Advanced programming in the UNIX environment*. Addison-Wesley, 2013.



RAYMOND E S. *The art of Unix programming*. Addison-Wesley, 2003.



MATTHEW N, STONES R. *Beginning linux programming*. John Wiley & Sons, 2008.



KERNIGHAN B W, RITCHIE D M. *The C programming language*. Prentice Hall, 2006.



KING K N. *C programming: a modern approach*. WW Norton & Company, 2008.



REEK K A. *Pointers on C*. Addison-Wesley, 1997.



WEISS M A, WEISS M A. *Data structures and algorithm analysis in C*. Benjamin/Cummings California, 1993.



WAITE M, PRATA S, MARTIN D. *C primer plus*. Sams, 1987.

Course Web Links

 <https://cs6.swfu.edu.cn/moodle>

 https://cs2.swfu.edu.cn/~wx672/lecture_notes/c/slides/

 https://cs2.swfu.edu.cn/~wx672/lecture_notes/c/src/

 <https://cs3.swfu.edu.cn/tech>

/etc/hosts

202.203.132.241 cs6.swfu.edu.cn

202.203.132.242 cs2.swfu.edu.cn

202.203.132.245 cs3.swfu.edu.cn

System Programming <https://github.com/angrave/SystemProgramming/wiki>

Beej's Guides <http://beej.us/guide/>

BLP4e <http://www.wrox.com/WileyCDA/WroxTitle/productCd-0470147628,descCd-DOWNLOAD.html>

TLPI <http://www.man7.org/tlpi/>

Weekly tech question

1. What was I trying to do?
2. How did I do it? (steps)
3. The expected output? The real output?
4. How did I try to solve it? (steps, books, web links)
5. How many hours did I struggle on it?

✉ wx672ster+linux@gmail.com

📖 Preferably in English

📖 in [stackoverflow](#) style

OR simply show me the tech questions you asked on any website



OVERSIMPLIFIED PROGRAMS
AHEAD!

1 Introduction

Program Languages

Machine code

The **binary numbers** that the CPUs can understand.

100111000011101111001111 ... and so on ...

Assembly language — friendly to humans

People don't think in numbers.

```
1  MOV  A,47  ;1010 1111
2  ADD  A,B   ;0011 0111
3  HALT      ;0111 0110
```

The ASM programs are translated to machine code by **assemblers**.

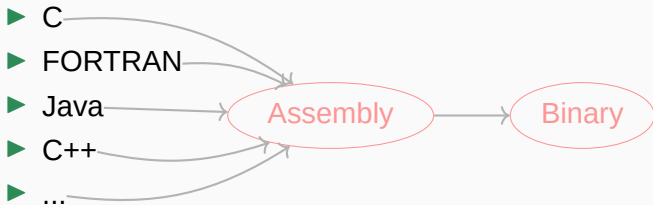
High level languages

Even easier to understand for humans. Examples:

- ▶ C
- ▶ FORTRAN
- ▶ Java
- ▶ C++
- ▶ ...

High level languages

Even easier to understand for humans. Examples:



Compilers do the translation work.

The History of C

- 1967 BCPL (Basic Computer Programming Language), Martin Richards
- 1970 B, Bell Labs, Ken Thompson
- 1970+ C, Bell Labs, Dennis Ritchie
- 1978 The C Programming Language, B.Kernighan/D.Ritchie
- 1980 C++, Bjarne Stroustrup
- 1989 ANSI C, American National Standards Institute
- 1999 ISO/IEC 9899 C, International Organisation for Standardization, 1999, the current Standard C
- 2000 C#, Anders Hejlsberg, Microsoft,

Hello, world!

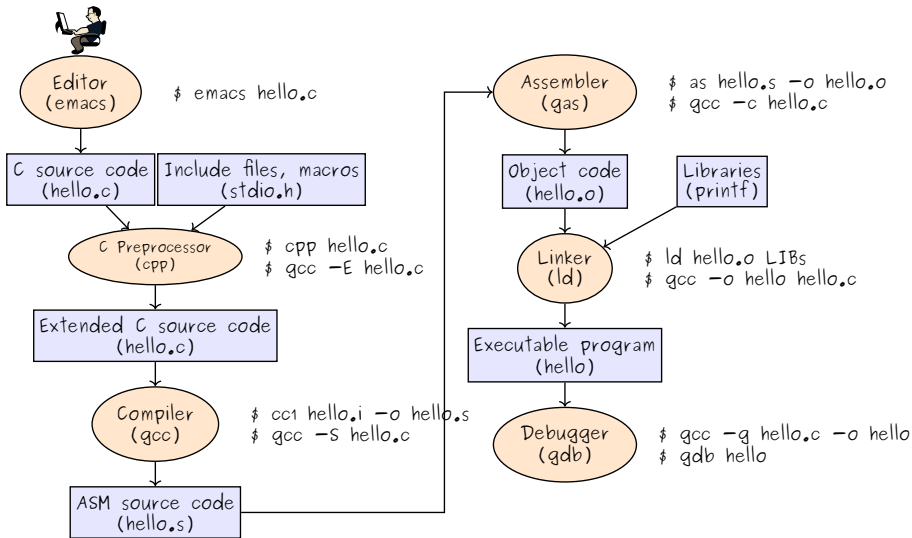
```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("Hello, world!\n");
6      return 0;
7  }
```

```
$ edit hello.c
```

```
$ gcc -Wall hello.c -o hello
```

```
$ ./hello
```

Toolchain



2 Basic Building Blocks of C

Basic Building Blocks of C

Data

different **types** of **variables**. Examples:

```
int v1;           int sum;           double i;  
int v2;           char c;
```

Instructions

Tell the computer what to do with the data.

- ▶ Operators (+, -, ×, ÷, ...)
- ▶ Assignment statement (=)
- ▶ Control statement (**if else**;
for; **while**; ...)

Examples:

```
1 | v1=5; v2=6;  
2 | sum = v1 + v2;  
3 | if (sum != 11) puts("Wrong!");
```

Operators for shortcuts

`x++;` `x += 2;` `x *= 4;` `x %= 6;`
`x--;` `x -= 3;` `x /= 5;`

```
1 | n = 5;  
2 | npp = n++; /* npp is 5 */  
3 | ppn = ++n; /* ppn is 6 */
```

The result (11 or 13) actually depends on the compiler

1. `int i=1;`
2. `i = (i++ * 5) + (i++ * 3);`

NO GOOD

$1 \xrightarrow{++} 2$
 $1 * 5 + 2 * 3 = 11$

$2 \xleftarrow{++} 1$
 $2 * 5 + 1 * 3 = 13$

Functions

```
1 int plus(int x, int y){  
2     int sum = x + y;  
3     return sum;  
4 }
```

```
1 int main(void){  
2     int v1=5, v2=6;  
3     int sum = plus(5,6);  
4     return 0;  
5 }
```

Recursion — A function calls itself

```
1 int factorial(int n){  
2     if (n == 0) return 1;  
3     return n*factorial(n-1);  
4 }
```

```
1 int main(void){  
2     return factorial(5);  
3 }
```


Files

Several files can be compiled together into a single executable

hello2.c

```
#include "hello.h"
```

```
int main(int argc, char *argv[]) {  
    if (argc != 2)  
        printf ("Usage: %s needs an argument.\n", argv[0]);  
    else  
        hi(argv[1]);  
    return 0;  
}
```

hello.h

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

```
int hi(char*);
```

hi.c

```
#include "hello.h"
```

```
int hi(char* s) {  
    printf ("Hello, %s\n", s);  
    return 0;  
}
```

Coding Style

```
1  /*****
2   * hello -- program to print out "Hello World".          *
3   *                                                         *
4   * Ralf Kaiser, September 2003                             *
5   *                                                         *
6   * Reference: Steve Oualline, Practical C Programming,    *
7   *               O'Reilly                                  *
8   *                                                         *
9   * Purpose: Demonstration of comments                     *
10  *                                                         *
11  *****/
12
13  #include <stdio.h>
14
15  int main(void)
16  {
17      /* Say Hello to the World */
18      printf("Hello World\n");
19      return 0;
20  }
```

Variable Types

Types char, int, float, double

Qualifiers short, long, long long, signed, unsigned

Type	Storage size	Value range
char	1 byte	$-2^7 \sim 2^7 - 1$ or $0 \sim 2^8 - 1$
signed char	1 byte	$-2^7 \sim 2^7 - 1$
unsigned char	1 byte	$0 \sim 2^8 - 1$
int	2 or 4 bytes	$-2^{15} \sim 2^{15} - 1$ or $-2^{31} \sim 2^{31} - 1$
unsigned int	2 or 4 bytes	$0 \sim 2^{16} - 1$ or $0 \sim 2^{32} - 1$
short	2 bytes	$-2^{15} \sim 2^{15} - 1$
unsigned short	2 bytes	$0 \sim 2^{16} - 1$
long	4 bytes	$-2^{31} \sim 2^{31} - 1$
unsigned long	4 bytes	$0 \sim 2^{32} - 1$

Integer

Platform dependent

```
1  #include <stdio.h>
2  #include <limits.h>
3
4  int main(void)
5  {
6      printf("Size of char: %ld\n", sizeof(char));
7      printf("Size of int: %ld\n", sizeof(int));
8      printf("Size of float: %ld\n", sizeof(float));
9      printf("Size of double: %ld\n", sizeof(double));
10     printf("short int: %ld\n", sizeof(short int));
11     printf("long int: %ld\n", sizeof(long int));
12     printf("unsigned long: %ld\n", sizeof(unsigned long int));
13     printf("long long: %ld\n", sizeof(long long int));
14     printf("unsigned long long: %ld\n", sizeof(unsigned long long int));
15     return 0;
16 }
```

Floating Point

Type	Size	Value range	Precision
float	4 byte	$1.2 \times 10^{-38} \sim 3.4 \times 10^{38}$	6 decimal places
double	8 byte	$2.3 \times 10^{-308} \sim 1.7 \times 10^{308}$	15 decimal places
long double	10 byte	$3.4 \times 10^{-4932} \sim 1.1 \times 10^{4932}$	19 decimal places

```
1  #include <stdio.h>
2  #include <float.h>
3  int main() {
4      printf("Size for float : %d \n", sizeof(float));
5      printf("Min float positive value: %E\n", FLT_MIN );
6      printf("Max float positive value: %E\n", FLT_MAX );
7      printf("Precision value: %d\n", FLT_DIG );
8      return 0;
9  }
```

Variable Names

- ✓ `int num_of_students = 10;`
- ✓ `int numOfStudents = 10;`
- ✓ `int _numOfStudents = 10;`
- ✓ `float pi = 3.14159;`
- ✓ `int sum=0, Sum=0, SUM=0; /* case sensitive*/`
- ✗ `3rd_entry /* starts with a number */`
- ✗ `all$done /* contains a '$'*/`
- ✗ `int /* reserved word */`
- ✗ `phone number /* has a space */`

Simple Operators

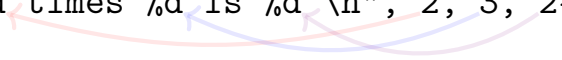
```
1  int term1, term2; /* 2 terms */
2  int sum;          /* sum of first and second term */
3  int diff;         /* difference of the two terms */
4  int modulo;       /* term1 modulus term2 */
5  int product;      /* term1 * term2 */
6  int ratio ;       /* term1 / term2 */
7
8  int main()
9  {
10     term1 = 1 + 2 * 4; /* 2*4=8, 8+1=9 */
11     term2 = (1 + 2) * 4; /* 1+2=3, 3*4=12 */
12     sum = term1 + term2; /* 9+12=21 */
13     diff = term1 - term2; /* 9-12=-3 */
14     modulo = term1 % term2; /* 9/12=0, remainder is 9 */
15     product = term1 * term2; /* 9*12=108 */
16     ratio = 9/12; /* 9/12=0 */
17     return(sum);
18 }
```

Floating Point vs. Integer Divide

Expression	Result	Result Type
19/10	1	integer
19.0/10	1.9	floating point
19.0/10.0	1.9	floating point


```
printf(format, expression1, expression2, ...)
```

```
printf("%d times %d is %d \n", 2, 3, 2*3);
```

A diagram illustrating argument passing in the printf function. Three curved arrows originate from the arguments '2', '3', and '2*3' in the function call and point to the corresponding format specifiers '%d', '%d', and '%d' in the format string. The first arrow is red, the second is blue, and the third is purple.

printf()

Escape Characters

Character	Name	Meaning
\b	backspace	move cursor one character to the left
\f	form feed	go to top of new page
\n	newline	go to the next line
\r	return	go to beginning of current line
\a	audible alert	'beep'
\t	tab	advance to next tab stop
\'	apostrophe	character '
\"	double quote	character "
\\	backslash	character \
\nnn		character number nnn (octal)

printf()

Format Statements

Conversion	Argument Type	Printed as
%d	integer	decimal number
%f	float	[-]m.dddddd (details below)
%X	integer	hex. number using A..F for 10..15
%c	char	single character
%s	char *	print characters from string until '\0'
%e	float	float in exp. form [-]m.dddddde xx
...

In addition,

%6d decimal integer, at least 6 characters wide

%8.2f float, at least 8 characters wide, two decimal digits

%.10s first 10 characters of a string

\$ man 3 printf

Arrays

```
1  #include <stdio.h>
2
3  float data[3];  /* data to average and total */
4  float total;    /* the total of the data items */
5  float average;  /* average of the items */
6
7  int main()
8  {
9      data[0] = 34.0;
10     data[1] = 27.0;
11     data[2] = 45.0;
12
13     total = data[0] + data[1] + data[2];
14     average = total / 3.0;
15     printf("Total %f Average %f\n", total, average);
16     return 0;
17 }
```

- ✓ `int data[3]={10,972,45};`
- ✓ `int data[]={10,972,45};`
- ✓ `int matrix[2][4]={{1,2,3,4},{10,20,30,40}};`

Strings

Strings are character arrays with the additional special character “\0” (NUL) at the end. E.g.:

```
char system[] = "Linux";
```

L	i	n	u	x	\0
---	---	---	---	---	----

The most common string functions

```
1 strcpy(string1, string2) /* copy string2 into string1 */
2 strcat(string1, string2) /* concatenate string2 onto
3                           the end of string1 */
4 length = strlen(string)  /* get the length of a string */
5 strcmp(string1, string2) /* 0 if string1 equals string2,
6                           otherwise nonzero */
```

Example

```
1  #include <string.h>
2  #include <stdio.h>
3
4  char first[100];      /* first name */
5  char last[100];       /* last name */
6  char full_name[200];  /* full name */
7
8  int main()
9  {
10     strcpy(first, "John"); /* Initialize first name */
11     strcpy(last, "Lennon"); /* Initialize last name */
12
13     strcpy(full_name, first); /* full = "John" */
14
15     strcat(full_name, " "); /* full = "John " */
16     strcat(full_name, last); /* full = "John Lennon" */
17
18     printf("The full name is %s\n", full_name);
19     return 0;
20 }
```

fgets()

Reading in strings from keyboard

```
char *fgets(char *s, int size, FILE *stream);
```

Example

```
1  #include <string.h>
2  #include <stdio.h>
3
4  char line[100]; /* Line we are looking at */
5
6  int main()
7  {
8      printf("Enter a line: ");
9      fgets(line, sizeof(line), stdin);
10
11     printf("The length of the line is: %d\n", strlen(line));
12     return 0;
13 }
```

```
$ man 3 fgets
```

Example

```
1  #include <stdio.h>
2  #include <string.h>
3
4  char first[100]; /* first name */
5  char last[100]; /* last name */
6  char full[200]; /* full name */
7
8  int main() {
9      printf("Enter first name: ");
10     fgets(first, sizeof(first), stdin);
11
12     printf("Enter last name: ");
13     fgets(last, sizeof(last), stdin);
14
15     strcpy(full, first);
16     strcat(full, " ");
17     strcat(full, last);
18
19     printf("The name is %s\n", full);
20     return 0;
21 }
```


scanf()

Reading in formatted input from stdin

```
int scanf(const char *format, ...);
```

Example

```
1  #include <stdio.h>
2
3  int main () {
4      char name[20];
5      int age;
6
7      printf("Enter name: ");
8      scanf("%s", name);
9      printf("Enter age: ");
10     scanf("%d", age);
11
12     printf("Your name is: %s\n", name);
13     printf("Your age is: %d\n", age);
14     return 0;
15 }
```

```
if ... else ...
```

```
1  #include<stdio.h>
2  int main(){
3      int a;
4
5      printf("Input an int: ");
6      scanf("%d", &a);
7
8      if( a != 10 ) printf("It\'s not 10.\n");
9
10     if( a < 10 )
11         printf("It\'s a small number.\n");
12
13     if( a > 10 ){
14         if( a < 20 )
15             printf("It\'s between 10 and 20.\n");
16         else if( a > 100 )
17             printf("It\'s larger than 100.\n");
18         else
19             printf("It\'s between 20 and 100.\n");
20     }
21     return a;
22 }
```

Relational Operators

< less than

<= less than or equal

= equal

> greater than

>= greater or equal than

!= not equal

Loops

while

```
1  #include<stdio.h>
2  int main(void)
3  {
4      int a = 0;
5      while( a < 10 ){
6          printf("a=%d\n", a);
7          a++;
8      }
9      return 0;
10 }
```

Loops

for

```
1  #include<stdio.h>
2  int main(void)
3  {
4      int a;
5      for( a=0; a<10; a++ ){
6          printf("a=%d\n", a);
7          a++;
8      }
9      return 0;
10 }
```

Loop Control Statements

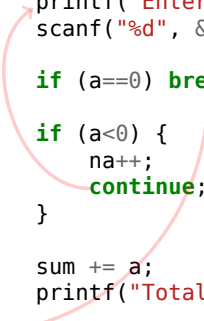
break

```
1  #include<stdio.h>
2  int main(void)
3  {
4      int a = 0;
5      while( a < 10 ){
6          printf("a=%d\n", a);
7          a++;
8          if (a>5) break;
9      }
10     return 0;
11 }
```

Loop Control Statements

continue

```
1  #include<stdio.h>
2  int main(void)
3  {
4      int a = 0, sum = 0, na=0;
5      while (1) {
6          printf("Enter # to add or 0 to stop: \n");
7          scanf("%d", &a);
8
9          if (a==0) break;
10
11         if (a<0) {
12             na++;
13             continue;
14         }
15
16         sum += a;
17         printf("Total: %d\n", sum);
18     }
19     printf("Final total %d ", sum);
20     printf("with %d negative items omitted.\n", na);
21     return 0;
22 }
```



switch

```
1  #include <stdio.h>
2
3  int main() {
4      char grade;
5      while(1){
6          printf("Input an uppercase letter: ");
7          scanf(" %c", &grade);/* try without the space */
8
9          switch(grade) {
10             case 'A' :
11                 printf("Excellent!\n");
12                 break;
13             case 'B' :
14             case 'C' :
15                 printf("Well done\n");
16                 break;
17             case 'D' :
18                 printf("You passed\n");
19                 break;
20             case 'F' :
21                 printf("Better try again\n");
22                 break;
23             default :
24                 printf("Invalid grade\n");
25             }
26         }
27         return 0;
28     }
```



```
1  switch (operator) {
2      case '+':
3          result += value;
4          break;
5      case '-':
6          result -= value;
7          break;
8      case '*':
9          result *= value;
10         break;
11     case '/':
12         if (value == 0) {
13             printf("Error:Divide by zero\n");
14             printf("    operation ignored\n");
15         } else
16             result /= value;
17         break;
18     default:
19         printf("Unknown operator %c\n", operator);
20         break;
21 }
```

3 The make Utility

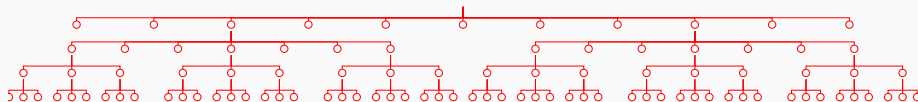
make

To compile a single C program:

```
$ gcc hello.c -o hello
```

OK. But...


What if you have a large project with 1000+ .c files?




Linux 4.9 source tree: 3799 directories, 55877 files

make: help you maintain your programs.

Makefile

```
1 target: dependencies
2 |  command
```

Example

```
1 hello: hello.c
2 |  gcc -o hello hello.c
```

```
$ info make makefiles
```

Makefile

```
1  edit: main.o kbd.o command.o display.o \  
2      insert.o search.o files.o utils.o  
3      gcc -Wall -o edit main.o kbd.o command.o display.o \  
4      insert.o search.o files.o utils.o  
5  
6  main.o: main.c defs.h  
7      gcc -c -Wall main.c  
8  kbd.o : kbd.c defs.h command.h  
9      gcc -c -Wall kbd.c  
10 command.o: command.c defs.h command.h  
11      gcc -c -Wall command.c  
12 display.o : display.c defs.h buffer.h  
13      gcc -c -Wall display.c  
14 insert.o: insert.c defs.h buffer.h  
15      gcc -c -Wall insert.c  
16 search.o: search.c defs.h buffer.h  
17      gcc -c -Wall search.c  
18 files.o: files.c defs.h buffer.h command.h  
19      gcc -c -Wall files.c  
20 utils.o: utils.c defs.h  
21      gcc -c -Wall utils.c  
22  
23 clean:  
24     rm edit main.o kbd.o command.o display.o \  
25     insert.o search.o files.o utils.o
```

./

- command.c
- display.c
- files.c
- insert.c
- kbd.c
- main.c
- search.c
- utils.c
- buffer.h
- command.h
- defs.h
- Makefile

4 C Concepts

The #include Instruction

```
1  #include <stdio.h>
2  #include "defs.h"
```

Header files: for keeping *definitions* and *function prototypes*. E.g.

- ▶ `#define SQR(x) ((x) * (x))`
- ▶ `ssize_t read(int fildes, void *buf, size_t nbyte);`

Standard header files: define data structures, macros, and function prototypes used by library routines, e.g. `printf()`.

```
$ ls /usr/include
```


Local include files: self-defined data structures, macros, and function prototypes.

```
$ gcc -E hello.c
```

The #define Instruction

Always put {} around all multi-statement macros!

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  #define DIE \
5      printf("Fatal Error! Abort\n"); exit(8);
6
7  int main(void)
8  {
9      int i = 1;
10     if (i<0) DIE
11     printf("Still alive!\n");
12     return 0;
13 }
```




```
1  #define DIE \
2      {printf("Fatal error! Abort\n"); exit(8);}
```



Why? gcc -E

Always put `()` around the parameters of a macro!

```
1  #include<stdio.h>
2
3  #define SQR(x) (x * x)
4  #define N 5
5
6  int main(void)
7  {
8      int i = 0;
9
10     for (i = 0; i < N; ++i) {
11         printf("x = %d, SQR(x) = %d\n", i+1, SQR(i+1));
12     }
13
14     return 0;
15 }
```



✓ `#define SQR(x) ((x) * (x))`

\$ gcc -E

Bitwise Operations

```
1  /*
2      7 6 5 4 3 2 1 0
3      ┌───┴───┐
4      │ E │ D │ B │ T │   │   │   │   │
5      └───┴───┘
6  */
7
8  char status;
9  status |= 0x40;      /* set 'D' bit */
10 if (status & 0x40);  /* test 'D' bit */
11 status &= ~0x40;     /* clear 'D' bit */
```

*E – error; D – done;
B – busy; T – trigger;
0x40 = 01000000b*

Pointers

```
c_ptr = &c;                                b_ptr = &b; a_ptr = &a;
```

↓ ↓ ↓

addr: 25ec 25ed 25ee 25ef | 25f0 25f1 25f2 | 25f3 | 25f4 25f5 25f6 25f7

							A				
3.1415926								1966			

float c;char b;int a;

Pointer Operators

`&` returns the **address** of a thing

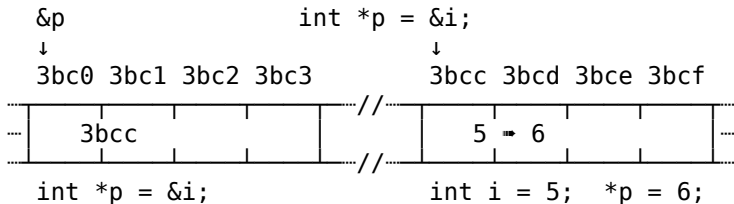
`*` return the **object (thing)** to which a pointer points at

```
int thing; int *thing_ptr;
```

C Code	Description
<code>thing</code>	the variable named 'thing'
<code>&thing</code>	address of 'thing' (a pointer)
<code>*thing</code>	✗
<code>thing_ptr</code>	pointer to an int
<code>*thing_ptr</code>	the int variable at the address <code>thing_ptr</code> points to
<code>&thing_ptr</code>	odd, a pointer to a pointer

Example

```
1  #include<stdio.h>
2
3  int main(void)
4  {
5      int i = 5;
6      int *p;
7      p = &i; /* now p pointing to i */
8      *p = 6; /* i = 6 */
9
10     printf("&i = %p, i = %d, *p = %d\n", &i, i, *p);
11     printf("&p = %p, p = %p\n", &p, p);
12     return 0;
13 }
```



Invalid operation

```
1  #include<stdio.h>
2
3  int main(void)
4  {
5      int i = 5;
6      printf("i = %d\n", *i); /* Wrong! */
7
8      return 0;
9  }
```

Invalid memory access

```
1  #include<stdio.h>
2
3  int main(void)
4  {
5      int *p = 5; /* should be (int *)5 */
6
7      printf("p = %p\n", p); /* p = 0x5 */
8      printf("&p = %p\n", &p); /* &p = 0x7ffda48a2068 */
9      printf("p = %c\n", *p); /* Invalid memory access */
10     return 0;
11 }
```

Call by Value

```
1  #include <stdio.h>
2  void inc_count(int count){
3      ++count;
4  }
5
6  int main(){
7      int count = 0;
8
9      while(count < 10){
10         inc_count(count);
11         printf("%d\n", count);
12     }
13
14     return 0;
15 }
```

Call by Value


```
1  #include <stdio.h>
2  void inc_count(int count){
3      ++count;
4  }
5
6  int main(){
7      int count = 0;
8
9      while(count < 10){
10         inc_count(count);
11         printf("%d\n", count);
12     }
13
14     return 0;
15 }
```



Call by value: only the **value** of 'count' is handed to the function `inc_count()`

Solution 1: return


```
1  #include <stdio.h>
2  int inc_count(int count){
3      return ++count;
4  }
5
6  int main(){
7      int count = 0;
8
9      while(count < 10){
10         count = inc_count(count);
11         printf("%d\n", count);
12     }
13
14     return 0;
15 }
```



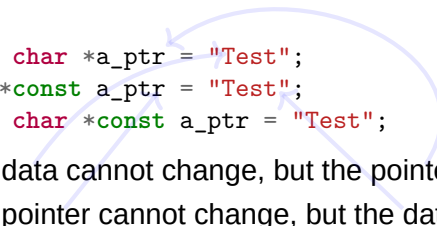
Pointers as Function Arguments

Solution 2: Call by reference

```
1  #include <stdio.h>
2  void inc_count(int *count_ptr){
3      ++(*count_ptr);
4  }
5
6  int main(){
7      int count = 0;
8
9      while(count < 10){
10         inc_count(&count);
11         printf("%d\n", count);
12     }
13
14     return 0;
15 }
```



const Pointers



```
1 | const char *a_ptr = "Test";  
2 | char *const a_ptr = "Test";  
3 | const char *const a_ptr = "Test";
```

The diagram consists of three lines of code. A blue oval encircles the first two lines. A blue arrow points from the pointer variable 'a_ptr' in line 1 to the string literal 'Test' in line 2. Another blue arrow points from the pointer variable 'a_ptr' in line 2 to the string literal 'Test' in line 3. A third blue arrow points from the pointer variable 'a_ptr' in line 3 to the string literal 'Test' in line 3.

1. The data cannot change, but the pointer can
2. The pointer cannot change, but the data it points to can
3. Neither can change

5 Pointers and Arrays

```

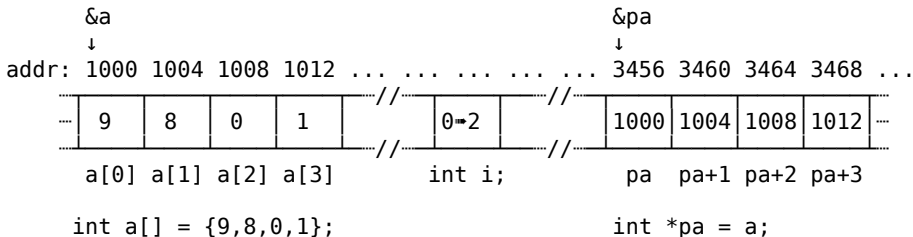
1  #include<stdio.h>
2
3  int main(void)
4  {
5      int a[] = {9,8,0,1};
6      int i = 0;
7
8      while (a[i] != 0)
9          ++i;
10
11     printf("ZERO at a[%d].\n", i);
12
13     return 0;
14 }

```

```

1  #include<stdio.h>
2
3  int main(void)
4  {
5      int a[] = {9,8,0,1};
6      int *pa = a;
7
8      while ((*pa) != 0)
9          ++pa;
10
11     printf("ZERO at a[%ld].\n", pa - a);
12     printf("pa = %p; a = %p\n", pa, a);
13     return 0;
14 }

```



Passing Arrays to Functions

When passing an array to a function, C will automatically change the array into a pointer.

```
1  #define MAX 10
2
3  void init_array_1(int a[]){
4      int i;
5
6      for (i = 0; i < MAX; ++i)
7          a[i] = 0;
8  }
9
10 void init_array_2(int *ptr){
11     int i;
12
13     for (i = 0; i < MAX; ++i)
14         *(ptr + i) = 0;
15 }
```

```
1  int main(void)
2  {
3      int array[MAX];
4
5      init_array_1(array);
6      init_array_1(&array[0]);
7      init_array_1(&array);
8      init_array_2(array);
9
10     return 0;
11 }
```

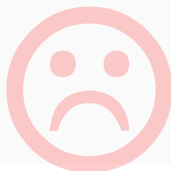
Arrays of Pointers

```
1  #include<stdio.h>
2
3  void print_msg(char *ptr_a[], int n) {
4      int i;
5      for (i = 0; i < n; i++)
6          printf(" %s", ptr_a[i]);
7
8      printf(".\n");
9  }
10
11 int main() {
12     char *message[9] =
13         {"Dennis", "Ritchie", "designed",
14          "the", "C", "language",
15          "in", "the", "1970s"};
16     print_msg(message, 9);
17     return 0;
18 }
```

How not to Use Pointers

Life is complicated enough, don't make it worse

```
1  /* Point to the first element of the array. */
2  data_ptr = &array[0];
3
4  /* Get element #0, data_ptr points to element #1. */
5  value = *data_ptr++;
6
7  /* Get element #2, data_ptr points to element #2. */
8  value = *++data_ptr;
9
10 /* Increment element #2, return its value.
11   Leave data_ptr alone. */
12 value = ++*data_ptr;
```



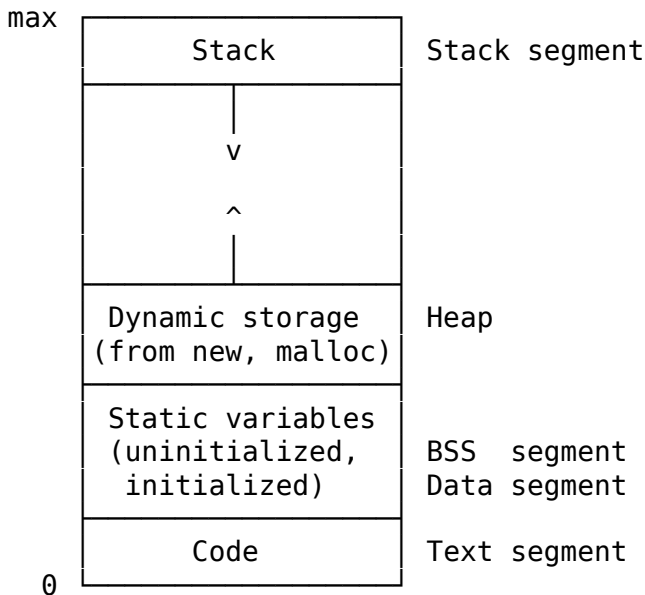
Just don't do it

```
1 void copy_string(char *p, char *q)
2 {
3     while (*p++ = *q++);
4 }
```



6 Memory Model

Memory Model



7 x86 Assembly

8 Hacker's Tools

9 Linux GUI Programming

9.1 ncurses

9.2 GTK

10 APUE

10.1 File I/O

10.2 Processes and Threads