[entry]none/global/
[entry]none/global/

C Programming under Linux

Lecture Handouts

WANG Xiaolin
wx672ster@gmail.com

December 5, 2017

Contents

List of Corrections

References

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.

```
MOV A,47 ;1010 1111
ADD A,B  ;0011 0111
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++
- ...

Compilers do the translation work.

Assembly
Binary

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!

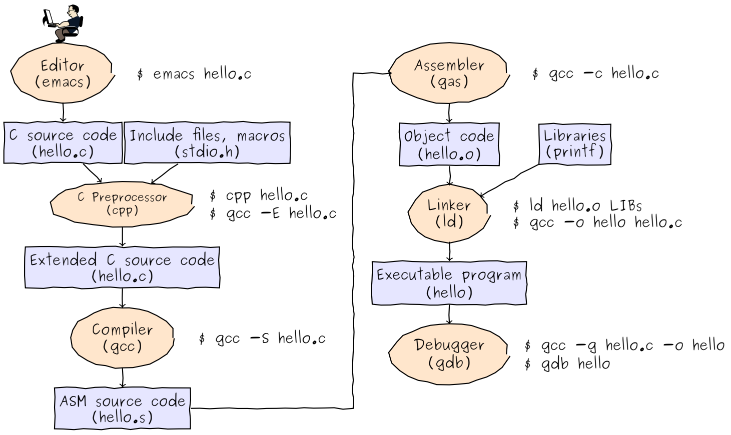
```c
#include <stdio.h>

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

```
$ edit hello.c
```

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

```
$ ./hello
```

Toolchain



Source code written by programmer in high-level language, in our case in `C`. We write c source code with a *text editor*, such as `emacs`, `vim`, etc.

Preprocessing is the first pass of any C compilation. It processes `include-files`, `conditional compilation instructions` and `macros`.

cpp The GNU C preprocessor

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

Compilation is the second pass. It takes the output of the preprocessor, and the `source code`, and generates `assembly source code`.

gcc/g++ GNU C/C++ compiler

```
$ gcc -S hello.c
```

Assembly is the third stage of compilation. It takes the assembly source code and produces an assembly listing with offsets. The assembler output is stored in an `object file`.

as the portable GNU assembler

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

Linking is the final stage of compilation. It combines object code with predefined routines from `libraries` and produces the `executable program`.

ld The GNU linker

```
$ gcc hello.c -lm
```

Wrapper The whole compilation process is usually not done 'by hand', but using a `wrapper` program that combines the functions of preprocessor(cpp), compiler(gcc/g++), assembler(as) and linker(ld).

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

See also: COMPILER, ASSEMBLER, LINKER AND LOADER: A BRIEF STORY.[1]

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 ($+, -, \times, \div, \ldots$)
- Assignment statement ($=$)
- Control statement (`if else; for; while; ...`)

Examples:

```
v1=5; v2=6;
sum = v1 + v2;
if (sum != 11) printf("Wrong!\n");
```

Operators for shortcuts

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

```
n = 5;
npp = n++; /* npp is 5 */
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);`

```
1 * 5 + 2 * 3 = 11

2 * 5 + 1 * 3 = 13
```

++␣++

NO GOOD

[1] http://www.tenouk.com/ModuleW.html

Functions

```
int plus(int x, int y){
  int sum = x + y;
  return sum;
}
```

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

Recursion — A function calls itself

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

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

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>
#include <stdlib.h>
int hi(char*);
```

hi.c

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

Coding Style

```
/*******************************************************
 * hello -- program to print out "Hello World".        *
 *                                                     *
 * Ralf Kaiser, September 2003                         *
 *                                                     *
 * Reference: Steve Oualline, Practical C Programming, *
 *             O'Reilly                                *
 *                                                     *
 * Purpose: Demonstration of comments                  *
 *                                                     *
 *******************************************************/

#include <stdio.h>

int main(void)
{
    /* Say Hello to the World */
    printf("Hello World\n");
    return 0;
}
```

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

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

int main(void)
{
    printf("Size of char: %ld\n", sizeof(char));
    printf("Size of int: %ld\n", sizeof(int));
    printf("Size of float: %ld\n", sizeof(float));
    printf("Size of double: %ld\n", sizeof(double));
    printf("short int: %ld\n", sizeof(short int));
    printf("long int: %ld\n", sizeof(long int));
    printf("unsigned long: %ld\n", sizeof(unsigned long int));
    printf("long long: %ld\n", sizeof(long long int));
    printf("unsigned long long: %ld\n", sizeof(unsigned long long int));
    return 0;
}
```

See also: **stevens2013advanced**.

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

```c
#include <stdio.h>
#include <float.h>
int main() {
    printf("Size for float : %d \n", sizeof(float));
    printf("Min float positive value: %E\n", FLT_MIN );
    printf("Max float positive value: %E\n", FLT_MAX );
    printf("Precision value: %d\n", FLT_DIG );
    return 0;
}
```

See also: C data types [2]

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

```c
int term1, term2; /* 2 terms */
int sum;          /* sum of first and second term */
int diff;         /* difference of the two terms */
int modulo;       /* term1 modulus term2 */
int product;      /* term1 term2 */
int ratio ;       /* term1 / term2 */

int main()
{
  term1 = 1 + 2 * 4;       /* 2*4=8, 8+1=9 */
  term2 = (1 + 2) * 4;     /* 1+2=3, 3*4=12 */
  sum = term1 + term2;     /* 9+12=21 */
  diff = term1 - term2;    /* 9-12=-3 */
  modulo = term1 % term2;  /* 9/12=0, remainder is 9 */
  product = term1 * term2; /* 9*12=108  */
  ratio = 9/12;            /* 9/12=0  */
  return(sum);
}
```

[2] https://www.tutorialspoint.com/cprogramming/c_data_types.htm

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);
```

`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

```c
#include <stdio.h>

float data[3];  /* data to average and total */
float total;    /* the total of the data items */
float average;  /* average of the items */

int main()
{
  data[0] = 34.0;
  data[1] = 27.0;
  data[2] = 45.0;

  total = data[0] + data[1] + data[2];
  average = total / 3.0;
  printf("Total %f Average %f\n", total, average);
  return 0;
}
```

- ✔ `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

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

Example

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

char first[100];     /* first name */
char last[100];      /* last name */
char full_name[200]; /* full name */

int main()
{
    strcpy(first, "John");  /* Initialize first name */
    strcpy(last, "Lennon"); /* Initialize last name */

    strcpy(full_name, first); /* full = "John" */

    strcat(full_name, " ");   /* full = "John " */
    strcat(full_name, last);  /* full = "John Lennon" */

    printf("The full name is %s\n", full_name);
    return 0;
}
```

fgets()

Reading in strings from keyboard

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

Example

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

char line[100]; /* Line we are looking at */

int main()
{
  printf("Enter a line: ");
  fgets(line, sizeof(line), stdin);

  printf("The length of the line is: %d\n", strlen(line));
  return 0;
}
```

```
$ man 3 fgets
```

Example

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

char first[100]; /* first name */
char  last[100]; /*  last name */
char  full[200]; /*  full name */

int main() {
    printf("Enter first name: ");
    fgets(first, sizeof(first), stdin);

    printf("Enter last name: ");
    fgets(last, sizeof(last), stdin);

    strcpy(full, first);
    strcat(full, " ");
    strcat(full, last);

    printf("The name is %s\n", full);
    return 0;
}
```

```
Output of fgets - Example 2:
kaiser@npl03:~/oreilly/pracc/full1> full1
Enter first name: John
Enter last name: Lennon
The name is John
 Lennon

What happened ? Why is the last name in a new line ?

     The fgets command gets the entire line, including the
     end-of-line. For example, "John" gets stored as
     {' J' ,' o' ,' h' ,' n' ,'  n' ,'  0' }.
     This can be fixed by using the statement
     first[strlen(first)-1] = '  0' ;
     which replaces the end-of-line with an end-of-string character
     and so end the string earlier.
```

scanf()

Reading in formatted input from stdin

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

Example

```c
#include <stdio.h>

int main () {
    char name[20];
    int age;

    printf("Enter name: ");
    scanf("%s", name);
    printf("Enter age: ");
    scanf("%d", age);

    printf("Your name is: %s\n", name);
    printf("Your age is: %d\n", age);
    return 0;
}
```

if ... else ...

```c
#include<stdio.h>
int main(){
    int a;

    printf("Input an int: ");
    scanf("%d", &a);

    if( a != 10 ) printf("It\'s not 10.\n");

    if( a < 10 )
        printf("It\'s a small number.\n");

    if( a > 10 ){
        if( a < 20 )
            printf("It\'s between 10 and 20.\n");
        else if( a > 100 )
            printf("It\'s larger than 100.\n");
        else
            printf("It\'s between 20 and 100.\n");
    }
    return a;
}
```

Relational Operators

<	less than	>	greater than
<=	less than or equal	>=	greater or equal than
==	equal	!=	not equal

Loops

while

```c
#include<stdio.h>
int main(void)
{
    int a = 0;
    while( a < 10 ){
        printf("a=%d\n", a);
        a++;
    }
    return 0;
}
```

Loops

for

```c
#include<stdio.h>
int main(void)
{
    int a;
    for( a=0; a<10; a++ ){
        printf("a=%d\n", a);
        a++;
    }
    return 0;
}
```

Loop Control Statements

break

```c
#include<stdio.h>
int main(void)
{
    int a = 0;
    while( a < 10 ){
        printf("a=%d\n", a);
        a++;
        if (a>5) break;
    }
    return 0;
}
```

Loop Control Statements

continue

```c
#include<stdio.h>
int main(void)
{
    int a = 0, sum = 0, na=0;
    while (1) {
        printf("Enter # to add or 0 to stop: \n");
        scanf("%d", &a);

        if (a==0) break;

        if (a<0) {
            na++;
            continue;
        }

        sum += a;
        printf("Total: %d\n", sum);
    }
    printf("Final total %d ", sum);
    printf("with %d negative items omitted.\n", na);
    return 0;
}
```

switch

```c
#include <stdio.h>

int main() {
    char grade;
    while(1){
        printf("Input an uppercase letter: ");
        scanf(" %c", &grade);/* try without the space */

        switch(grade) {
        case 'A' :
            printf("Excellent!\n");
            break;
        case 'B' :
        case 'C' :
            printf("Well done\n");
            break;
        case 'D' :
            printf("You passed\n");
            break;
        case 'F' :
            printf("Better try again\n");
            break;
        default :
            printf("Invalid grade\n");
        }
    }
    return 0;
}
```

```c
switch (operator) {
 case '+':
     result += value;
     break;
 case '-':
     result -= value;
     break;
 case '*':
     result *= value;
     break;
 case '/':
     if (value == 0) {
         printf("Error:Divide by zero\n");
         printf("   operation ignored\n");
     } else
         result /= value;
     break;
 default:
     printf("Unknown operator %c\n", operator);
     break;
 }
```

3 The make Utility

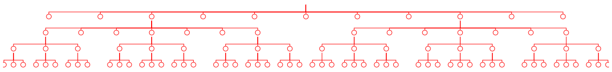
make

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

```
target: dependencies
│──TAB──→ command
```

Example

```
hello: hello.c
│──TAB──→ gcc -o hello hello.c
```

```
$ info make makefiles
```

Makefile

```
edit: main.o kbd.o command.o display.o \
                insert.o search.o files.o utils.o
        gcc -Wall -o edit main.o kbd.o command.o display.o \
                insert.o search.o files.o utils.o

main.o: main.c defs.h
        gcc -c -Wall main.c
kbd.o : kbd.c defs.h command.h
        gcc -c -Wall kbd.c
command.o: command.c defs.h command.h
        gcc -c -Wall command.c
display.o : display.c defs.h buffer.h
        gcc -c -Wall display.c
insert.o: insert.c defs.h buffer.h
        gcc -c -Wall insert.c
search.o: search.c defs.h buffer.h
        gcc -c -Wall search.c
files.o: files.c defs.h buffer.h command.h
        gcc -c -Wall files.c
utils.o: utils.c defs.h
        gcc -c -Wall utils.c

clean:
        rm edit main.o kbd.o command.o display.o \
           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

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

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

#define DIE \
    printf("Fatal Error! Abort\n"); exit(8);

int main(void)
{
    int i = 1;
    if (i<0) DIE
    printf("Still alive!\n");
    return 0;
}
```

✗

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

Why? `gcc -E`

✓

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

```
#include<stdio.h>

#define SQR(x) (x * x)
#define N 5

int main(void)
{
    int i = 0;

    for (i = 0; i < N; ++i) {
        printf("x = %d, SQR(x) = %d\n", i+1, SQR(i+1));
    }

    return 0;
}
```

✗

✔ *#define SQR(x) ((x) * (x))*

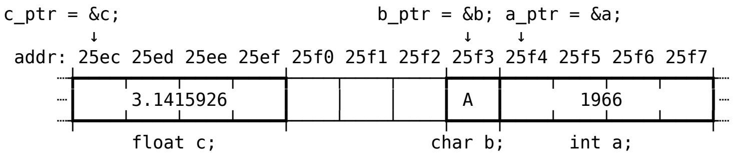
```
$ gcc -E
```

Bitwise Operations

```
/*
   7 6 5 4 3 2 1 0
  +-+-+-+-+-+-+-+-+   E — error; D — done;
  |E|D|B|T| | | | |   B — busy;   T — trigger;
  +-+-+-+-+-+-+-+-+   0x40 = 01000000b
*/

char status;
status |= 0x40;      /*   set 'D' bit */
if (status & 0x40); /*  test 'D' bit */
status &= ~0x40;     /* clear 'D' bit */
```

Pointers

```c
#include<stdio.h>

int main(void)
{
    int a = 1966;
    char b = 'A';
    float c = 3.1415926;
    int   *a_ptr = &a; /* a pointer to an integer */
    char  *b_ptr = &b; /* a pointer to a  char    */
    float *c_ptr = &c; /* a pointer to a  float   */

    printf("&a = %p, sizeof(a) = %ld\n", a_ptr, sizeof(a));
    printf("&b = %p, sizeof(b) = %ld\n", b_ptr, sizeof(b));
    printf("&c = %p, sizeof(c) = %ld\n", c_ptr, sizeof(c));
    return 0;
}
```



Pointer Operators

& returns the *address* of a thing

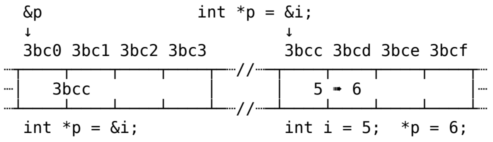
∗ return the *object (thing)* to which a pointer points at

`int thing; int *thing_ptr;`

C Code	Description
`thing`	the variable named ‘thing’
`&thing`	address of ‘thing’ (a pointer)
`*thing`	✘
`thing_ptr`	pointer to an int
`*thing_ptr`	the int variable at the address `thing_ptr` points to
`&thing_ptr`	odd, a pointer to a pointer

Example

```c
#include<stdio.h>

int main(void)
{
    int i = 5;
    int *p;
    p = &i; /* now p pointing to i */
    *p = 6; /* i = 6 */

    printf("&i = %p, i = %d, *p = %d\n", &i, i, *p);
    printf("&p = %p, p = %p\n", &p, p);
    return 0;
}
```



Invalid operation

```c
#include<stdio.h>

int main(void)
{
    int i = 5;
    printf("*i = %d\n", *i); /* Wrong! */

    return 0;
}
```

This is trying to treat the value of i as an memory address. But the memory address 5 is not accessible by this program.

Invalid memory access

```c
#include<stdio.h>

int main(void)
{
    int *p = 5; /* should be (int *)5 */

    printf(" p = %p\n",  p); /*  p = 0x5 */
    printf("&p = %p\n", &p); /* &p = 0x7ffda48a2068 */
    printf("*p = %c\n", *p); /* Invalid memory access */
    return 0;
}
```

This is trying to treat the value of p as an memory address. But the memory address 5 is not accessible by this program.

Call by Value

```c
#include <stdio.h>
void inc_count(int count){
    ++count;
}

int main(){
    int count = 0;

    while(count < 10){
        inc_count(count);
        printf("%d\n", count);
    }

    return 0;
}
```

✗

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

Solution 1: return

```c
#include <stdio.h>
int inc_count(int count){
    return ++count;
}

int main(){
    int count = 0;

    while(count < 10){
        count = inc_count(count);
        printf("%d\n", count);
    }

    return 0;
}
```

✓

1. read the *value* of count, and pass it to inc_count();
2. inc_count() uses the *value* of count to do the calculations;
3. return the result to main().

Pointers as Function Arguments

Solution 2: Call by reference

```c
#include <stdio.h>
void inc_count(int *count_ptr){
    ++(*count_ptr);
}

int main(){
    int count = 0;

    while(count < 10){
        inc_count(&count);
        printf("%d\n", count);
    }

    return 0;
}
```

1. pass the address of count to `inc_count()`;
2. `inc_count()` operates directly on count.

This is more efficient than solution 1 (Imagining you are operating on a large data structure rather than an `int`).

`const` Pointers

```c
const char *a_ptr = "Test";
char *const a_ptr = "Test";
const char *const a_ptr = "Test";
```

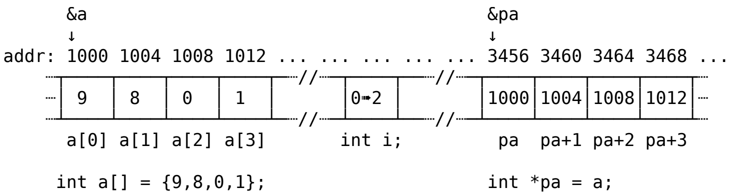
1. The data cannot change, but the pointer can
2. The pointer cannot change, but the data it points to can
3. Neither can change

5 Pointers and Arrays

```c
#include<stdio.h>

int main(void)
{
  int a[] = {9,8,0,1};
  int i = 0;

  while (a[i] != 0)
    ++i;

  printf("ZERO at a[%d].\n", i);

  return 0;
}
```

```c
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 }
```

C automatically scales pointer arithmetic so that it works correctly, by incrementing/decrementing by the correct number of bytes. For example, at line 11, the value of pa is 1008, and the value of a is 1000. But the result of "pa - a" is 2 rather than 8. This means pa is *two ints* ahead of a.



Passing Arrays to Functions

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

```
#define MAX 10

void init_array_1(int a[]){
  int i;

  for (i = 0; i < MAX; ++i)
      a[i] = 0;
}

void init_array_2(int *ptr){
  int i;

  for (i = 0; i < MAX; ++i)
      *(ptr + i) = 0;
}
```

```
int main(void)
{
  int array[MAX];

  init_array_1(array);
  init_array_1(&array[0]);
  init_array_1(&array);
  init_array_2(array);

  return 0;
}
```

Arrays of Pointers

```
#include<stdio.h>

void print_msg(char *ptr_a[], int n) {
  int i;
  for (i = 0; i < n; i++)
      printf(" %s", ptr_a[i]);

  printf(".\n");
}

int main() {
  char *message[9] =
      {"Dennis", "Ritchie", "designed",
       "the", "C", "language",
       "in", "the", "1970s"};
  print_msg(message, 9);
  return 0;
}
```

Once you've declared an array, you can't reassign it. Why? [https://stackoverflow.com/questions/17077505/string-pointer-and-array-of-chars-in-c]

Consider an assignment like

```
char *my_str = "foo"; // Declare and initialize a char pointer.
my_str = "bar"; // Change its value.
```

The first line declares a char pointer and "aims" it at the first letter in foo. Since foo is a string constant, it resides somewhere in memory with all the other constants. When you reassign the pointer, you're assigning a new value to it: the address of bar. But the original string, foo, remains unchanged. You've moved the pointer, but haven't altered the data.

When you declare an array, however, you aren't declaring a pointer at all. You're reserving a certain amount of memory and giving it a name. So the line

```
char c[5] = "data";
```

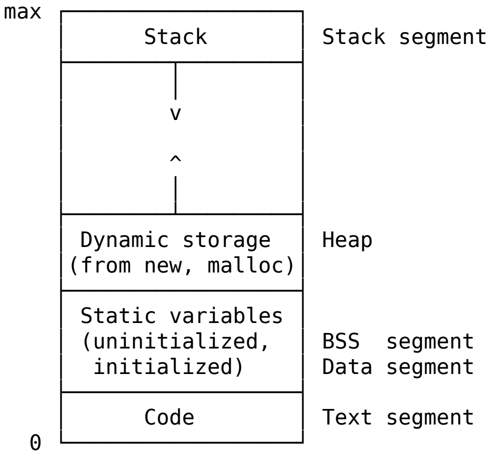
starts with the string constant data, then allocates 5 new bytes, calls them c, and copies the string into them. You can access the elements of the array exactly as if you'd declared a pointer to them; arrays and pointers are (for most purposes) interchangeable in that way.

But since arrays are not pointers, you cannot reassign them. You can't make c "point" anywhere else, because it's not a pointer; it's the name of an area of memory. For example,

```
char c[5] = "data";
char b[5] = "beta";
b = c; /* Wrong! 'b[]' cannot be reassigned (pointing to elsewhere). */
```

6 Memory Model

Memory Model



- See also: **erickson2008hacking**
- stack setup [linux sys slides]
- `http://www.dirac.org/linux/gdb/02a-Memory_Layout_And_The_Stack.php`
- gdb (info frame, info args, info locals, ...)
- TODO: make a good example using both printf() and gdb to show internals of a process

TODO!

7 x86 Assembly

- **erickson2008hacking**
- **jeff16assembly**
- **neveln2000linux**

8 Hacker's Tools

gdb, objdump, readelf, nm, ...

- **levine2000linkers**
- **salomon1992assemblers**

9 Linux GUI Programming

9.1 ncurses

9.2 GTK

10 APUE

10.1 File I/O

10.2 Processes and Threads