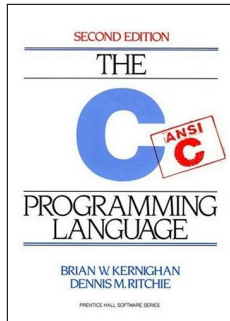


A Shotgun Introduction to C

Stephen A. Edwards

Columbia University

Fall 2013



History

Adding Two Numbers

Variables, Types, and Constants

Operators

If-then-else, switch, while, and for

Arrays and Strings

Structs

Functions

Pointers

Pointers and Structs

Strlen: An Example

C History

Developed between 1969 and 1973
along with Unix

Due mostly to Dennis Ritchie

Designed for systems programming

- ▶ Operating systems
- ▶ Utility programs
- ▶ Compilers
- ▶ Filters

Evolved from B, which evolved from BCPL



C History

Original machine, a DEC PDP-11, was very small:

24K bytes of memory, 12K used for operating system

Written when computers were big, capital equipment

Group would get one, develop new language, OS



Adding Two Numbers

```
int add()      /* Function that returns an integer */
{
    int x, y, z; /* Variables x, y, and z are integers */
    x = 38;     /* Set x to 38 */
    y = 4;     /* Set y to 4 */
    z = x + y;  /* Set z to the sum of x and y */
    return z;   /* Return z as the result of add() */
}
```

End statements with semicolons

Text between /* and */ is ignored (a comment)

Programs are mostly function definitions and global variables.

Variables

Names must start with a letter; may contain letters, numbers, and underscores.

```
a A a_variable aVariable a50 ex 12_ /* OK */  
two-words 42_is_the_answer /* BAD */
```

Must be declared before they're used

```
int a, b, c; /* 32-bit signed binary integers */  
char c, d; /* Single letter, digit, etc. */  
  
a = 42;  
b = 18;  
f = 3; /* BAD: f not declared */  
c = 'o';  
d = '#';  
q = '4'; /* BAD: q not declared */
```

Constants

```
#define ROWS 10  
#define COLUMNS 40  
  
pos = y * COLUMNS + rows;
```

This turns into

```
pos = y * 40 + rows;
```

The “#” must be in the leftmost column.

Common Operators

```
int a, b, c;
```

```
a = b + c;    /* Addition */
```

```
a = b - c;    /* Subtraction */
```

```
a = -(b + c); /* Negation */
```

```
a = b * c;    /* Multiplication */
```

```
a = b / c;    /* Division (integer result) */
```

```
a = b % c;    /* Remainder (modulus) */
```

```
a = b < c;    /* a is non-zero if b is less than c */
```

```
a = b > c;    /* non-zero if b is greater than c */
```

```
a = b <= c;   /* b less than or equal to c */
```

```
a = b >= c;   /* b greater than or equal to c */
```

```
a = b == c;   /* a is non-zero if b is equal to c */
```

```
a = b != c;   /* a is non-zero if b different than c */
```


Assignment Operators

A convenient shorthand:

```
a += 3;    /* Increase a by 3 */
```

```
a = a + 3; /* Equivalent */
```

```
b *= 2;    /* Double b */
```

```
b = b * 2; /* Equivalent */
```

Most operators have assignment variants.

Bitwise Operators

Internally, numbers represented in binary.

$$\begin{aligned}10100101_2 &= 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + \\ &\quad 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 128 + 32 + 4 + 1 \\ &= 165_{10}\end{aligned}$$

Bitwise operators work directly on bits:

$$\begin{array}{r} \text{AND: } \begin{array}{r} 10110 \\ \& 10011 \\ \hline 10010 \end{array} \quad \text{OR: } \begin{array}{r} 10110 \\ | 10011 \\ \hline 10111 \end{array} \quad \text{XOR: } \begin{array}{r} 10110 \\ \wedge 10011 \\ \hline 00101 \end{array} \end{array}$$

Increment/Decrement Operators

```
a = a + 1; /* Common operation */
a += 1;   /* One shorthand */
a++;     /* Even more succinct */

for (i = 0 ; i < 10 ; i++) { /* Very common idiom */
    /* i = 0, 1, 2, ..., 9 */
}

a = 3;
b = a++; /* Postincrement: means b = 3; a = 4; */
b = ++a; /* Preincrement:  means a = 5; b = 5; */

a = 3;
b = a--; /* Postdecrement: means b = 3; a = 2; */
b = --a; /* Predecrement:  means a = 1; b = 1; */
```

The If-Else Statement

```
if (a == 3)
    c = 2;          /* Runs if a is 3. One statement: braces optional */

if (b == 4 && c == 2) { /* && is logical AND */
    c = 5;          /* Runs if b is 4 and c is 2 */
    a = a + 3;
}                  /* Two statements: braces mandatory */

if (a > b) {
    c = 1;          /* Runs if a is greater than b */
} else {
    c = 5;          /* Runs if a is not greater than b */
}

if (a > b || c == 3) { /* || is logical OR */
    c = 5;          /* Runs if a is greater than b or c is 3 */
} else {
    a = b + 2;
}
}
```

The Switch Statement: A Multiway Conditional

```
switch (a + 1) {  
  case 2:  
    c = 8; /* Runs if a is 1 */  
    b = 2;  
    break;  
  
  case 0:  
  case 1: /* Multiple cases allowed */  
    b = 3; /* Runs if a is -1 or 0 */  
    break;  
  
  case 42: /* Case labels need not be contiguous */  
    c = 12;  
    /* No break: falls through to next case! */  
  
  case 4:  
    c = 15; /* Runs if a is 3 or 41 */  
    break;  
  
  default: /* a default is optional */  
    c = 0; /* Runs if no other case matches */  
    break; /* Good style */  
}
```

The While Statement

```
int gcd(int a, int b)
{
    while (a != b) { /* Repeat while a and b are different */

        if (a > b)
            a -= b; /* a is larger; subtract b from it */
        else
            b -= a; /* b is larger; subtract a from it */

    }
    return a;
}
```

The For Statement

```
/* Sum the numbers from 1 to n */  
int sumup(int n)  
{  
    int i, s;  
    i = 0;  
    s = 0;  
    while (i <= n) {  
        s += i;  
        i += 1;  
    }  
    return s;  
}
```

```
/* Sum the numbers from 1 to n */  
int sumup(int n)  
{  
    int i, s;  
    s = 0;  
    for (i = 0 ; i <= n ; i += 1)  
        s += i;  
    return s;  
}
```

Arrays

```
int i;
int a[10];           /* Array of 10 integers */
int b[] = { 2,3,7,6 }; /* Initial values */

a[0] = 3;
a[2] = 5;
a[9] = 18;
a[10] = 42; /* BAD: only a[0] ... a[9] */
a[-1] = 2;  /* BAD: positive indexes only */

a[1] = b[0]; /* a[0] = 2 */
a[3] = b[1]; /* a[3] = 3 */
b[3] = 42;
b[4] = 18;   /* BAD: only b[0] .. b[3] */

i = 5;
a[i] = 42;   /* a[5] = 42; */
i = 4;
a[i] = 10;   /* a[4] = 10; */
```


Strings

```
/* Strings are null-terminated arrays of characters */  
  
char name1[] = "Stephen";  
/* is equivalent to */  
char name2[] = {'S', 't', 'e', 'p', 'h', 'e', 'n', 0};  
  
name1[5] = 'a';  
  
/* name1 now "Stephan" */
```

Structs

```
struct point { /* Define an aggregate type "struct point" */
    int x;
    int y;
};

struct point p; /* Declare a new point */

p.x = 10; /* Set its coordinates */
p.y = 15;

printf("(%d,%d)\n", p.x, p.y);

struct point q = { 320, 200 }; /* Initialize contents */

p = q; /* Copy one point to another */

struct rect {
    struct point southwest;
    struct point northeast;
};

struct rect r;
r.southwest.x = 10;
r.southwest.y = 5;
r.northeast.x = 125;
r.northeast.y = 200;
```

Functions

```
int num_calls = 0; /* global variable */

int power(int base, int n)
{
    int p; /* Different than main's p */

    for ( p = 1 ; n > 0 ; --n )
        p *= base;

    num_calls++;

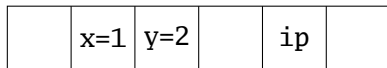
    return p;
}

int main() /* main function always runs first */
{
    int n, p;

    n = power(2, 5); /* n = 32 */
    p = power(3, 3); /* p = 27 */

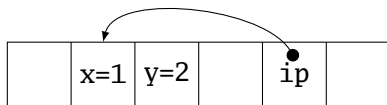
    p = num_calls; /* p = 2 */
}
```

Pointers



```
int x = 1, y = 2;  
int *ip;
```

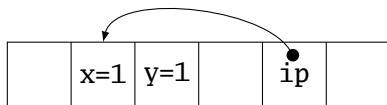
Pointers



```
int x = 1, y = 2;  
int *ip;
```

```
ip = &x;
```

Pointers

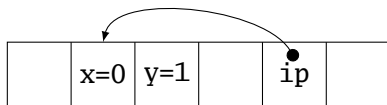


```
int x = 1, y = 2;  
int *ip;
```

```
ip = &x;
```

```
y = *ip;
```

Pointers



```
int x = 1, y = 2;  
int *ip;
```

```
ip = &x;
```

```
y = *ip;
```

```
*ip = 0;
```

Pointers

```
void swap(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

Does this work?

Pointers

```
void swap(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

Does this work?
Nope.

```
void swap(int *px, int *py)
{
    int temp;

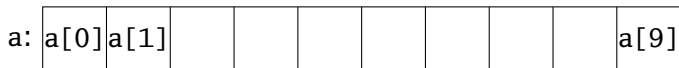
    temp = *px; /* get data at px */
    *px = *py; /* get data at py */
    *py = temp; /* write data at py */
}

void main()
{
    int a = 1, b = 2;

    /* Pass addresses of a and b */
    swap(&a, &b);

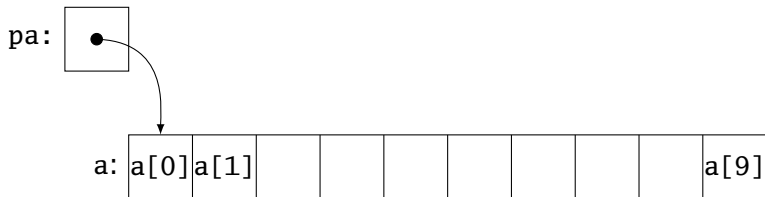
    /* a = 2 and b = 1 */
}
```

Arrays and Pointers



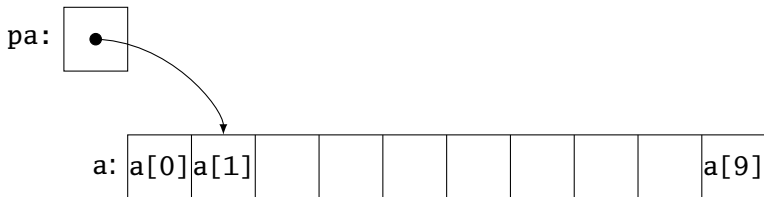
```
int a[10];
```

Arrays and Pointers



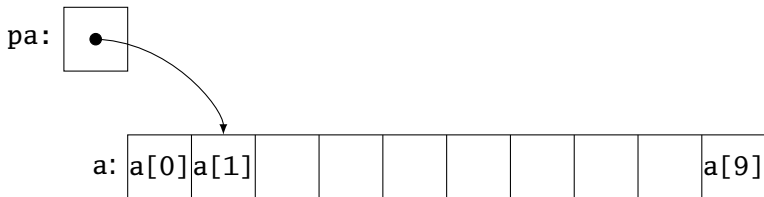
```
int a[10];  
int *pa = &a[0];
```

Arrays and Pointers



```
int a[10];  
int *pa = &a[0];  
pa = pa + 1;
```

Arrays and Pointers

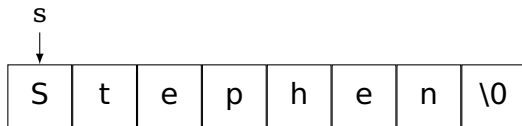


```
int a[10];  
int *pa = &a[0];  
pa = pa + 1;  
pa = &a[1];
```

Pointers and Structs

```
struct point {  
    int x;  
    int y;  
};  
  
struct point p, *pp;  
  
p.x = 100;  
p.y = 200;  
  
pp = &p;          /* pp now points to p */  
  
(*pp).x = 50;    /* Assign to x field of p */  
pp->x = 50;       /* Equivalent */  
pp->y = 42;
```

Strlen: An Example



```
int strlen(const char *s)
{
    int n;

    for (n = 0 ; *s != '\0' ; s++)
        n++;

    return n;
}

void main()
{
    char ste[] = "Stephen";
    int l = strlen(ste);
}
```

Separate Compilation

file1.c

```
extern void bar();
char a[] = "Hello";

int main() {
    bar();
}

void baz(char *s) {
    printf("%\%%s", s);
}
```

file2.c

```
extern char a[];
extern void baz(char *);

static char b[6];

void bar() {
    strcpy(b, a);
    baz(b);
}
```


Better Style: Header Files

myfiles.h

```
#ifndef _MYFILES_H
#define _MYFILES_H

/* in file1.c */
extern void bar();
extern char a[];

/* in file2.c */
extern void
baz(char *);

#endif
```

file1.c

```
#include "myfiles.h"

char a[] = "Hello";

int main() {
    bar();
}

void baz(char *s) {
    printf("%\%%s", s);
}
```

file2.c

```
#include "myfiles.h"

static char b[6];

void bar() {
    strcpy(b, a);
    baz(b);
}
```