
C Programming Language Review and Dissection III

Lecture 5



Today

Pointers

Strings

Formatted Text Output

Reading Assignment:

- Patt & Patel “Pointers and Arrays”
 - Chapter 16 in 2nd edition
 - Chapter 17 in 1st edition and online notes

Pointers

A *pointer* variable holds the *address* of the data, rather than the *data* itself

To make a pointer point to variable **a**, we can specify the *address* of **a**

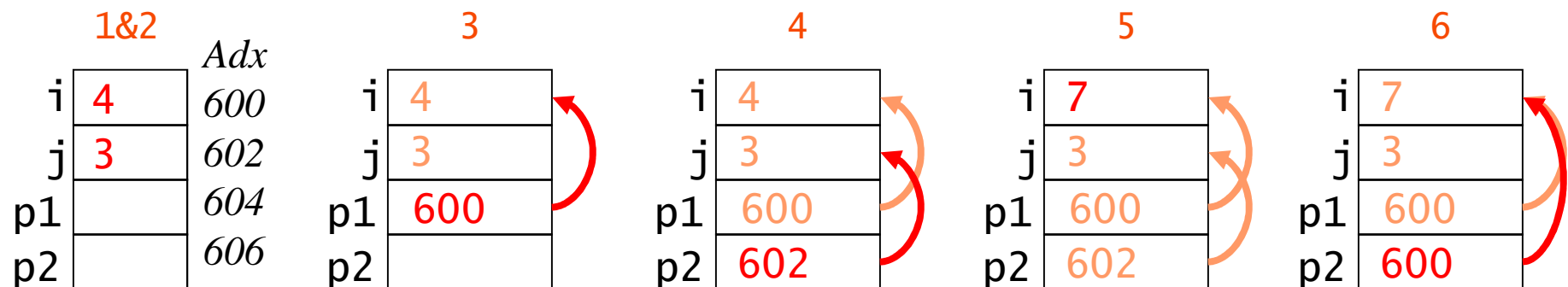
- address operator **&**

The data is accessed by *dereferencing* (following) the pointer

- indirection operator ***** works for reads and writes

Assigning a new value to a pointer variable changes *where the variable **points**, not the data*

```
void main ( ) {  
    int i, j;  
    int *p1, *p2;  
1   i = 4;  
2   j = 3;  
3   p1 = &i;  
4   p2 = &j;  
5   *p1 = *p1+*p2;  
6   p2 = p1;  
}
```



More about Pointers

Incrementing and decrementing pointers to array elements

- Increment operator ++ makes pointer advance to next element (next larger address)
- Decrement operator -- makes pointer move to previous element (next smaller address)
- These use the size of the variable's base type (e.g. int, char, float) to determine what to add
 - **p1++** corresponds to **p1 = p1 + sizeof(int)**;
 - sizeof is C macro which returns size of type in bytes

```
int a[18];
int * p;
p = &a[5];
*p = 5; /* a[5]=5 */
p++;
*p = 7; /* a[6]=7 */
p--;
*p = 3; /* a[5]=3 */
```

Pre and post

- Putting the ++/-- **before** the pointer causes inc/dec **before** pointer is used
 - int *p=100, *p2;
 - **p2 = ++p**; assigns **102** to integer pointer **p2**, and **p** is **102** afterwards
- Putting the ++/-- **after** the pointer causes inc/dec **after** pointer is used
 - char *q=200, *q2;
 - **q2 = q--**; assigns **200** to character pointer **q2**, and **q** is **199** afterwards

What else are pointers used for?

Data structures which reference each other

- lists
- trees
- etc.

Exchanging information between procedures

- Passing arguments (e.g. a structure) quickly – just pass a pointer
- Returning a structure

Accessing elements within arrays (e.g. string)

Pointers and the M16C ISA

Address space of M16C is 1 megabyte

- Need 20 bits to address this

This space is divided into two areas

- Near: 64 kilobytes from 00000h to 0FFFFh can be addressed with a 16-bit pointer (top 4 bits of 20-bit address are 0)
 - Pointer is shorter (2 bytes)
 - Pointer operations are faster
 - Note: internal RAM and SFRs are in this space
- Far: Entire 1 megabyte area from 00000h to FFFFFh can be addressed with a 20-bit pointer
 - Pointer is longer (4 bytes used (1.5 bytes wasted!))
 - Pointer operations are slower, since ALU operates on 16 bits at a time

Details in section 2.3 of M16C C Programming Manual

Specifying Areas

By default, RAM data is near, ROM data is far

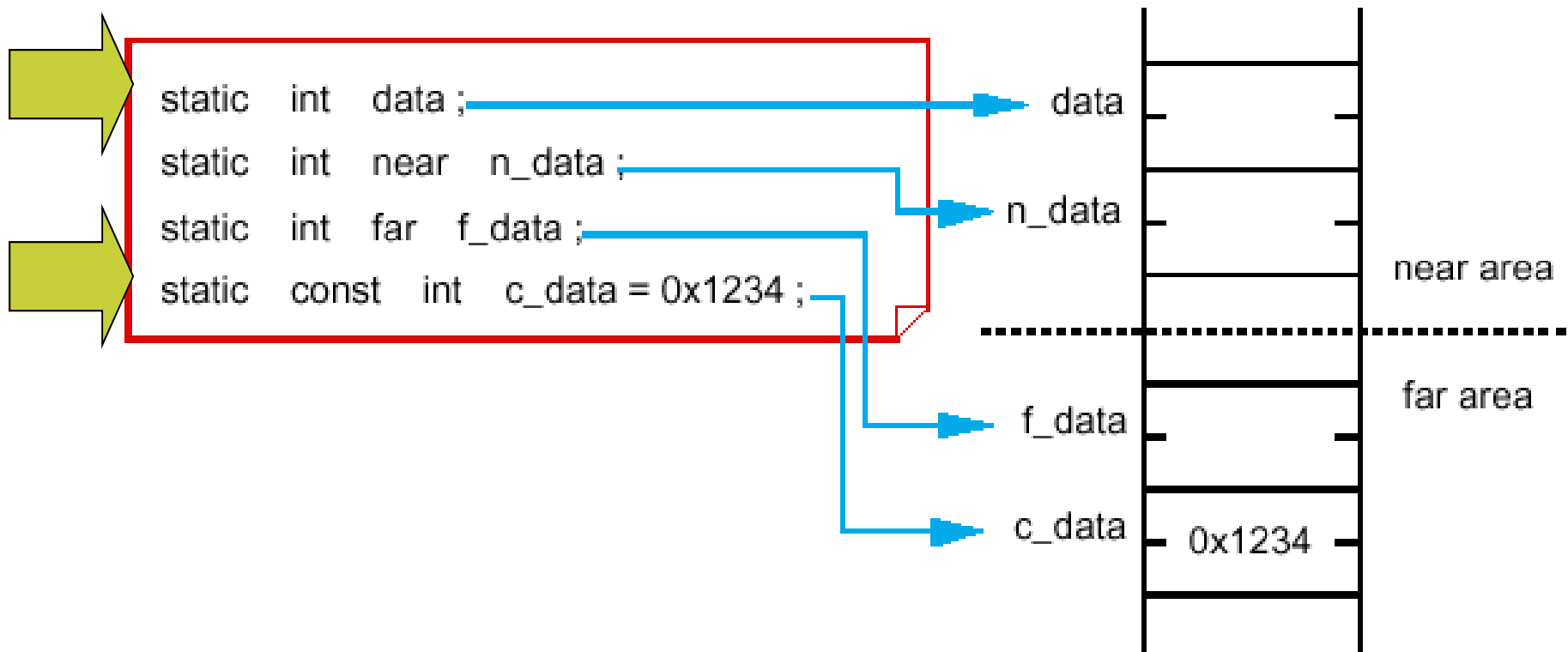


Figure 2.3.1 near/far of static variables

Pointers and the M16C ISA (II)

Default locations

- Near area: RAM data
 - data, bss
- Far area: ROM data
 - rom, program
 - const data

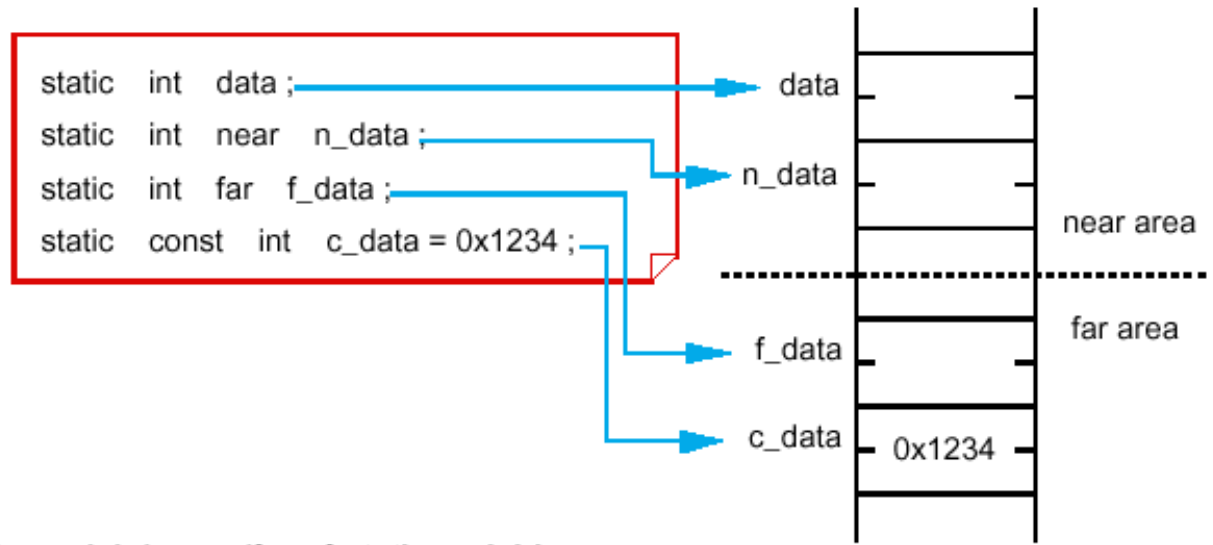


Figure 2.3.1 near/far of static variables

Pointer sizes chosen by compiler based on area holding type of data

- Near pointer (16 bits) used for near data
- Far pointer (32 bits) used for far data

NC30 doesn't recognize near/far keywords?

- `int * near near_data;` *does not compile*
- `int * far far_data;` *does not compile*
- `int near near_data;` *does not compile*
- `int far far_data;` *does not compile*

Strings

See Section 16.3.4 of Patt & Patel.

There is no “string” type in C.

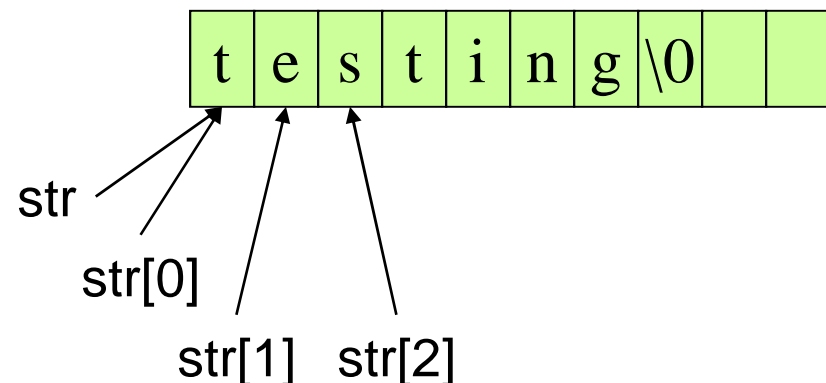
Instead an **array of characters** is used - *char a[44]*

The string is terminated by a NULL character (value of 0, represented in C by `\0`).

- Need an extra array element to store this null

Example

- `char str[10] = “testing”;`



Formatted String Creation

Common family of functions defined in `stdio.h`

- `printf`: print to standard output
- `sprintf`: print to a string
- `fprintf`: print to a file

Syntax: `sprintf(char *str, char * frmt, arg1, arg2, arg3 ..);`

- `str`: destination
- `fmt`: format specifying what to print and how to interpret arguments
 - `%d`: signed decimal integer
 - `%f`: floating point
 - `%x`: unsigned hexadecimal integer
 - `%c`: one character
 - `%s`: null-terminated string
- `arg1`, etc: arguments to be converted according to format string

printf Examples – strings and integers

```
char s1[30], s2[30];
```

```
int a=5, b=10, c=-30;
```

```
char ch='$';
```

```
printf(s1, "Testing");
```

s1

Testing

```
printf(s2, "a=%d, b=%d", a, b);
```

s2

a=5, b=10

```
printf(s1, "b=%x, c=%d", b, c);
```

s1

b=a, c=-30

```
printf(s1, "b=0x%x", b);
```

s1

b=0xa

```
printf(s2, "s1=%s", s1);
```

s2

s1=b=0xa

```
printf(s1, "%c %c", ch, s2);
```

s1

\$ s

sprintf Examples – floating-point

Variation on %f format specifier

– %-w.pf

- - = left-justify. Optional
- w = minimum field width (# of symbols)
- p = precision (digits after decimal point)

Examples

```
float f1=3.14, f2=9.991, f3=-19110.331;
```

```
char s1[30], s2[30];
```

```
sprintf(s1, "%f", f1);
```

```
sprintf(s1, "%f", f3);
```

```
sprintf(s1, "%4.1f", f2);
```

s1

3.140000

s1

-19110.3

s1

10.0

sprintf Examples – More Integers

Variation on %d format specifier for integers (d/i/o/x/u)

– %-w.pd

- - = left justify. Optional
- w = minimum field width (# of symbols)
- p = precision (digits). Zero pad as needed

Examples

```
int a=442, b=1, c=-11;  
char s1[30], s2[30];  
sprintf(s1, "%5d", a);
```

s1
442

```
sprintf(s1, "%-4d", b);
```

s1
1

```
sprintf(s1, "%4d", b);
```

s1
1

```
sprintf(s1, "%-5.4d", c);
```

s1
-011

String Operations in string.h

Copy **ct** to **s** including terminating null character. Returns a pointer to **s**.

```
– char* strcpy(char* s, const char* ct);  
  s1 = “cheese”;  
  s2 = “limburger”;  
  strcpy(s1, s2); /* s1 = limburger */
```

Concatenate the characters of **ct** to **s**. Terminate **s** with the null character and return a pointer to it.

```
– char* strcat(char* s, const char* ct);  
  s1 = “cheese”;  
  s2 = “ puffs”;  
  strcat(s1, s2); /* s1 = cheese puffs */
```

More String Operations

Concatenate at most **n** characters of **ct** to **s**. Terminate **s** with the null character and return a pointer to it.

```
– char* strncat(char* s, const char* ct, int n);  
  s1 = “cheese”;  
  s2 = “ puffs”;  
  strncat(s1, s2, 4); /* cheese puf */
```

Compares two strings. The comparison stops on reaching a **null** terminator. Returns a 0 if the two strings are identical, less than zero if **s2** is greater than **s1**, and greater than zero if **s1** is greater than **s2**. (Alphabetical sorting by ASCII codes)

```
– int strcmp(const char* s1, const char* s2);  
  s1 = “cheese”;  
  s2 = “chases”;  
  strcmp(s1,s2); /* returns non-zero number */  
  strcmp(s1, “cheese”); /* returns zero */
```

More String Operations

Return pointer to first occurrence of **c** in **s1**, or **NULL** if not found.

```
– char* strchr(const char* s1, int c);  
  s1 = “Smeago1 and Deago1”;  
  char a *;  
  a = strchr(s1, “g”); /* returns pointer to s1[4] */
```

Return pointer to last occurrence of **c** in **s1**, or **NULL** if not found.

```
– char* strrchr(const char* s1, int c);  
  s1 = “Smeago1 and Deago1”;  
  char a *;  
  a = strrchr(s1, “a”); /* returns pointer to s1[14] */
```

Can use the returned pointer for other purposes

```
*a = ‘\0’; /* s1 = “Smeago1 and De” */  
strcat(s1, “spair”); /* s1 = “Smeago1 and Despair” */
```