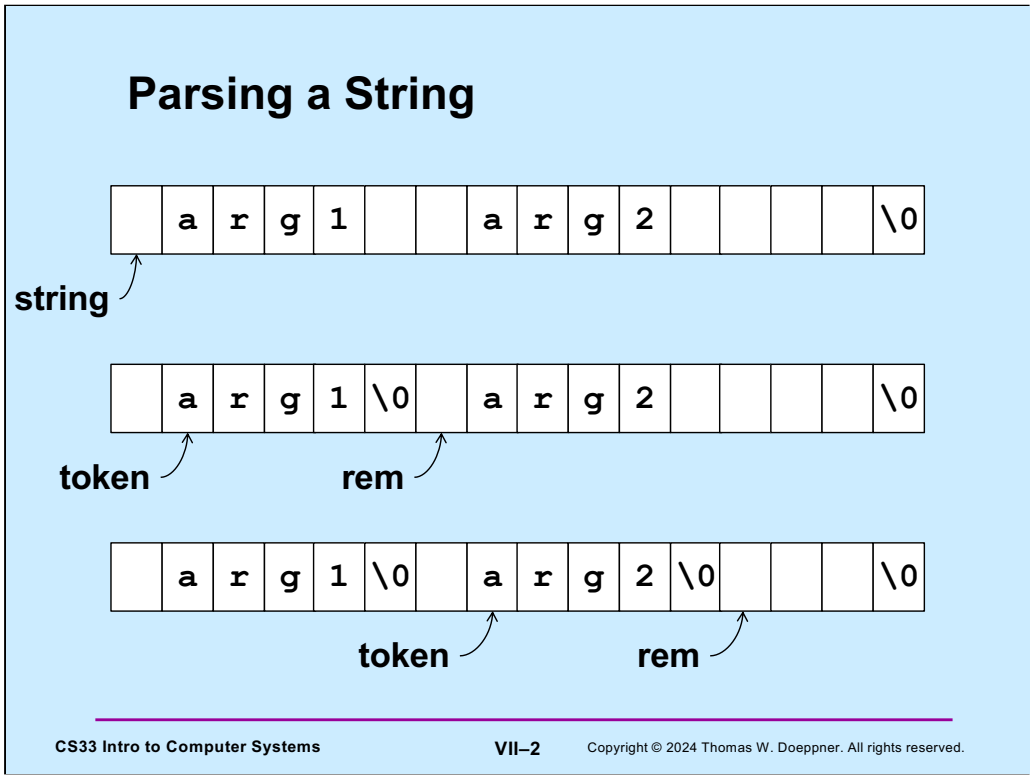
CS 33

Introduction to C
Part 7

Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective." 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated "Supplied by CMU" in the notes section of the slides.



Suppose we have a string of characters (perhaps typed into the command line of a shell). We'd like to parse this string to pull out individual words or "tokens" (to be used as arguments to a command); these tokens are separated by one or more characters of white space. Starting with a pointer to this string, we call a function that null-terminates the first token and returns a pointer to that word (**token**) and sets **rem** to point to the remainder of the string. We call it again to get the second token, etc.

Designing the Parse Function

- It modifies the string being parsed
  - puts nulls at the end of each token
- Each call returns a pointer to the next token
  - how does it know where it left off the last time?
    - » how is *rem* dealt with?

The parse function must keep track of where it left off after each call to it. One way of doing this is via the use of a static local variable.

Design of *strtok*

- **char** *strtok(**char** *string,
  const char *sep)
  - **if *string* is non-NULL, *strtok* returns a pointer to the first token in *string* (and keeps track of where the next token would be)**
  - **if *string* is NULL, *strtok* returns a pointer to the token just after the one returned in the previous call, or NULL if there are no more tokens**
  - **tokens are separated by any non-empty combination of characters in *sep***

strtok is a standard function in the C strings library. Note that, since the second argument is declared to be a pointer to a constant, there's a promise that **strtok** will not modify what its second argument points to.

Using *strtok*

```
int main() {
  char line[] = "  arg0   arg1 arg2   arg3   ";
  char *str = line;
  char *token;
  while ((token = strtok(str, " \t\n")) != NULL) {
    printf("%s\n", token);
    str = NULL;
  }
  return 0;
}
```

Output:

```
arg0
arg1
arg2
arg3
```

strtok Code part 1

```
char *strtok(char *string, const char *sep) {
  static char *rem = NULL;
  if (string == NULL) {
    if (rem == NULL) return NULL;
    string = rem;
  }
  int len = strlen(string);
  int slen = strspn(string, sep);
      // initial separators
  if (slen == len) {
    // string is all separators
    rem = NULL;
    return NULL;
  }
```


Note the static declaration of **rem** – this allows **strtok** to keep track of the remaining portion of the string.

strtok Code part 2

```
  string = &string[slen]; // skip over separators
  len -= slen;
  int tlen = strcspn(string, sep); // length of first token
  if (tlen < len) {
    // token ends before end of string: terminate it with 0
    string[tlen] = '\0';
    rem = &string[tlen+1];
  } else {
    // there's nothing after this token
    rem = NULL;
  }
  return string;
}
```


Numeric Conversions

```
short a;
int b;
float c;

b = a;    /* always works */
a = b;    /* sometimes works */
c = b;    /* sort of works */
b = c;    /* sometimes works */
```


Assigning a short to an int will always work, since all possible values of a short can be represented by an int. The reverse doesn't always work, since there are many more values an int can take on than can be represented by a short.

A float can represent an int in the sense that the smallest and largest ints fall well within the range of the smallest (most negative) and largest floats. However, floats have fewer significant digits than do ints and thus, when converting from an int to a float, there may well be a loss of precision.

When converting from a float to an int there will not be any loss of precision, but large floats and small (most negative) floats cannot be represented by ints.

Implicit Conversions (1)

```
float x, y=2.0;
int i=1, j=2;

x = i/j + y;
  /* what's the value of x? */
```


x's value will be 2, since the result of the (integer) division of **i** by **j** will be 0.

Implicit Conversions (2)

```
float x, y=2.0;
int i=1, j=2;
float a, b;

a = i;
b = j;
x = a/b + y;
  /* now what's the value of x? */
```


Here the values of **i** and **j** are converted to float before being assigned to **a** and **b**, thus the value assigned to **x** is 2.5.

Explicit Conversions: Casts

```
float x, y=2.0;
int i=1, j=2;

x = (float)i/(float)j + y;
  /* and now what's the value of x? */
```


Here we do the int-to-float conversion explicitly; **x**'s value will be 2.5.

Purposes of Casts

- **Coercion**

```
int i, j;
float a;
a = (float)i/(float)j;
```

modify the value appropriately

- **Intimidation**

```
float x, y;
    // sizeof(float) == 4
swap((int *)&x, (int *)&y);
```

it's ok as is (trust me!)

"Coercion" is a commonly accepted term for one use of casts. "Intimidation" is not. The concept is more commonly known as a "sidecast". Coercion means to convert something of one datatype to another. Intimidation (or sidecasting) means to treat an instance one datatype as being another datatype without doing any conversion of the actual data. Intimidation works only for pointer datatypes.

Quiz 1

- **Will this work?**

```
double x, y; //sizeof(double) == 8

...
swap((int *)&x, (int *)&y);
```

a) **yes**
b) **no**

Caveat Emptor

- **Casts tell the C compiler: "Shut up, I know what I'm doing!"**
- **Sometimes true**

```
float x, y;
swap((int *)&x, (int *)&y);
```

- **Sometimes false**

```
double x, y;
swap((int *)&x, (int *)&y);
```


The call to **swap** makes sense as long as what **x** and **y** point to are the same size as **int**'s.

The moral is to be careful with casting, particularly intimidation casts, since they effectively turn off type checking.

Nothing, and More …

- ***void*** **means, literally, nothing:**

```
void NotMuch(void) {
   printf("I return nothing\n");
}
```

- **What does *void* * mean?**
  - **it's a pointer to anything you feel like**
    - » **a generic pointer**

The **void *** type is an exception to the rule that the type of the target of a pointer must be known.

Rules

- **Use with other pointers**

```
int *x;
void *y;
x = y; /* legal */
y = x; /* legal */
```

- **Dereferencing**

```
void *z;
func(*z); /* illegal!*/
func(*(int *)z); /* legal */
```


Dereferencing a pointer must result in a value with a useful type. “void” is not a useful type.

Swap, Revisited

```
void swap(int *i, int *j) {
  int tmp;
  tmp = *j; *j = *i; *i = tmp;
}
/* can we make this generic? */
```


Can we write a version of **swap** that handles a variety of data types?

An Application: Generic Swap

```
void gswap (void *p1, void *p2,
      int size) {
  int i;
  for (i=0; i < size; i++) {
      char tmp;
      tmp = ((char *)p1)[i];
      ((char *)p1)[i] = ((char *)p2)[i];
      ((char *)p2)[i] = tmp;
  }
}
```


Note that there is a function in the C library that one may use to copy arbitrary amounts of data — it's called **memmove**. To see its documentation, use the Linux command "man memmove".

Using Generic Swap

```
short a=1, b=2;
gswap(&a, &b, sizeof(short));

int x=6, y=7;
gswap(&x, &y, sizeof(int));

int A[] = {1, 2, 3}, B[] = {7, 8, 9};
gswap(A, B, sizeof(A));
```


Fun with Functions (1)

```
void ArrayDouble(int A[], int len) {
  int i;
  for (i=0; i<len; i++)
      A[i] = 2*A[i];
}
```


Fun with Functions (2)

```
void ArrayBop(int A[],
     int len,
     int (*func)(int)) {
  int i;
  for (i=0; i<len; i++)
     A[i] = (*func)(A[i]);
}
```


Here **func** is declared to be a pointer to a function that takes an **int** as an argument and returns an **int**.

What's the difference between a pointer to a function and a function? A pointer to a function is, of course, the address of the function. The function itself is the code comprising the function. Thus, strictly speaking, if **func** is the name assigned to a function, **func** really represents the address of the function. You might think that we should invoke the function by saying "***func**", but it's understood that this is what we mean when we say "**func**". Thus, when one calls **ArrayBop**, one supplies the name of the desired function as the third argument, without prepending "&".

Fun with Functions (3)

```
int triple(int arg) {
  return 3*arg;
}

int main() {
  int A[20];
  … /* initialize A */
  ArrayBop(A, 20, triple);
  return 0;
}
```


Here we define another function that takes a single **int** and returns an **int**, and pass it to **ArrayBop**.

typedef

- Allows one to create new names for existing types

```
typedef int *IntP_t;
```

```
IntP_t x;
```

– means the same as

```
int *x;
```


More typedefs

```
typedef struct complex {
  float real;
  float imag;
} complex_t;

complex_t i, *ip;
```


A standard convention for C is that names of datatypes end with "_t". Note that it's not necessary to give the struct a name in this example (we could have omitted the "complex" following "struct"). It's also not necessary for the name of the type to be different from the name of the struct. Though it's a bit confusing, we could have coded the above as:

```
typedef struct complex {
            float real;
            float imag;
} complex;


complex i, *ip;
```

After doing this, "struct complex" and "complex" would mean exactly the same thing.

Not a Quiz

- **What's A?**

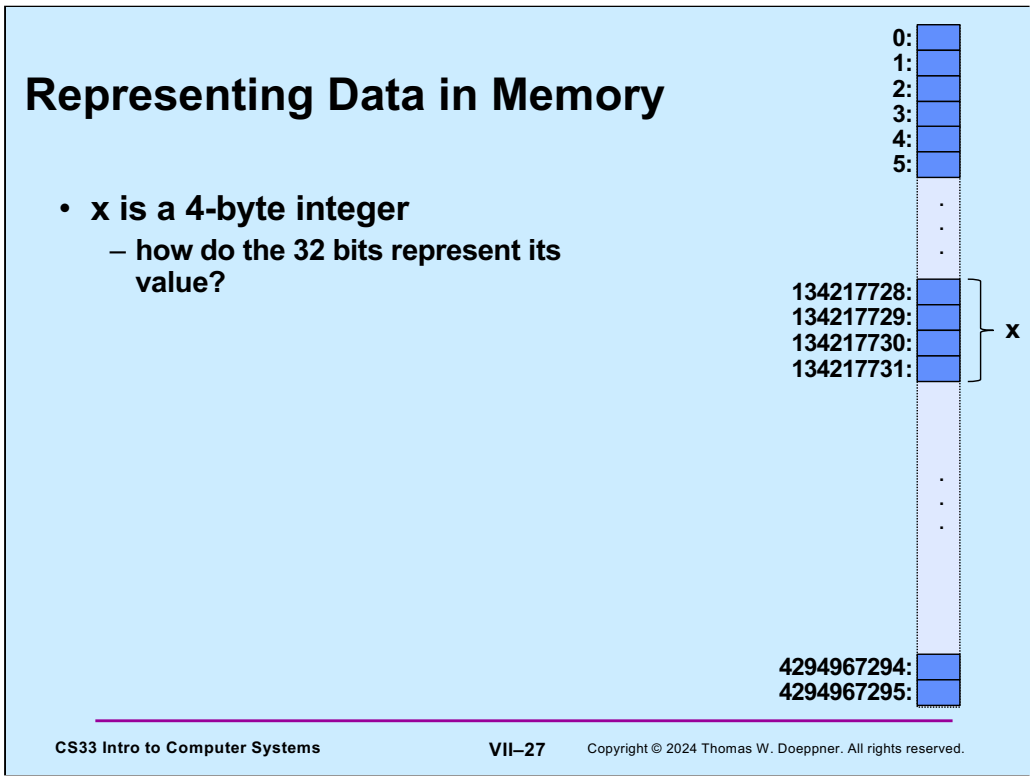
```
typedef double X_t[N];
X_t A[M];
```

a) an array of M doubles
b) an MxN array of doubles
c) an NxM array of doubles
d) a syntax error

CS 33

Data Representation, Part 1

Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective." 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated "Supplied by CMU" in the notes section of the slides.



In the diagram, x is an int occupying bytes 134217728, 134217729, 134217730, and 134217731. Its address is 134217728; its size is 4 (bytes).

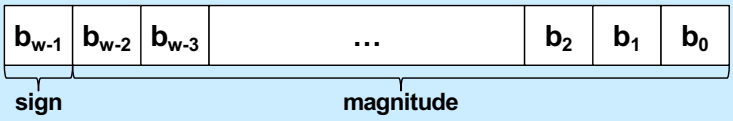
Unsigned Integers

$$\text{value} = \sum_{i=0}^{w-1} b_i \cdot 2^i$$

If a computer word is to be interpreted as an unsigned integer, we can do so as shown in the slide, where w is the number of bits in the word.

Signed Integers

- **Sign-magnitude**



$$\text{value} = (-1)^{b_{w-1}} \cdot \sum_{i=0}^{w-2} b_i \cdot 2^i$$

- two representations of zero!
  - computer must have two sets of instructions
    - one for signed arithmetic, one for unsigned

We might also want to interpret the contents of a computer word as a signed integer. There are a few options for how to do this. One straightforward approach is shown in the slide, where we use the high-order (leftmost) bit as the "sign bit": 0 means positive and 1 means negative. However, this has the somewhat weird result that there are two representations of zero. This further means that the computer would have to have two implementations of arithmetic instructions: one for signed arithmetic, the other for unsigned arithmetic.

Signed Integers

- **Ones' complement**
  - **negate a number by forming its bit-wise complement**
    - » **e.g., (-1)·01101011 = 10010100**

b_{w-1} = 0 ⇒ non-negative number

value = $\sum_{i=0}^{w-2} b_i \cdot 2^i$

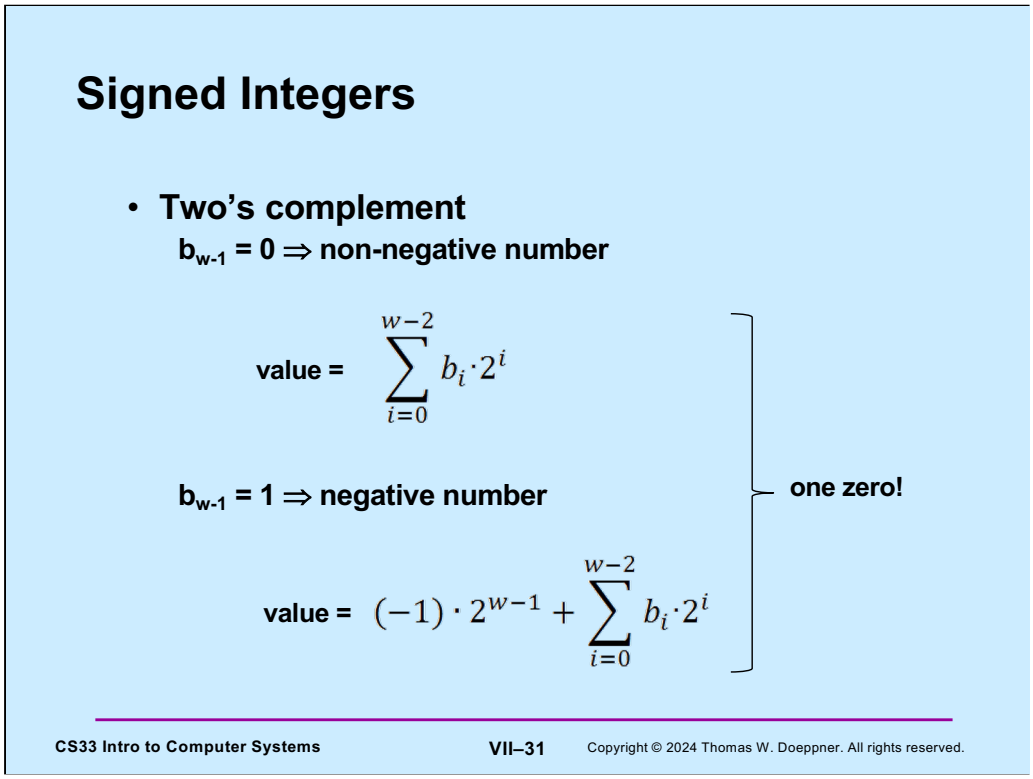
b_{w-1} = 1 ⇒ negative number

value = $\sum_{i=0}^{w-2} (b_i\text{-}1) \cdot 2^i$

two zeros!

In ones' complement, a number is positive if its leftmost bit is zero negative otherwise. We negate a number by complementing **all** its bits. Thus, if the leftmost bit is zero, a one in position **i** of the remaining bits contributes a value of $\mathbf{2^i}$ and a zero contributes nothing. But if the leftmost bit is one, a zero in position **i** contributes a value of $\mathbf{-2^i}$ and a one contributes nothing.

Note that the most-significant bit serves as the sign bit. But, as with sign-magnitude, the computer would need two sets of instructions: one for signed arithmetic and one for unsigned.



There's only one zero!

Two's complement is used on pretty much all of today's computers to represent signed integers.

Note that the high-order (most-significant) bit represents -2^{w-1}. All the other bits represent positive numbers.

Example

- **w = 4**

```
0000: 0        1000: -8
0001: 1        1001: -7
0010: 2        1010: -6
0011: 3        1011: -5
0100: 4        1100: -4
0101: 5        1101: -3
0110: 6        1110: -2
0111: 7        1111: -1
```

Signed Integers

- **Negating two's complement**

$$value = -b_{w-1}2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i$$

  - **how to compute *–value*?**
    (~value)+1

To negate a two's-complement number, simply complement each of its bits, then add one to the result. We show why this works in the next slide.

Signed Integers

- **Negating two's complement (continued)**

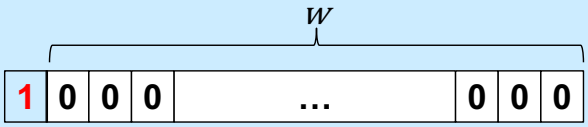
$$value + (\sim value + 1)$$

$$= (value + \sim value) + 1$$

$$= (2^w - 1) + 1$$

$$= 2^w$$

$$=$$



If we add to the two's complement representation of a w-bit number the result of adding one to its bitwise complement, we get a w+1-bit number whose low-order w bits are zeroes and whose high-order bit is one. However, since we're constrained to only w bits, the result is a w-bit value of all zeroes, plus an overflow. If we ignore the overflow, the result is zero.

Quiz 2

- We have a computer with 4-bit words that uses two's complement to represent signed integers. What is the result of subtracting 0010 (2) from 0001 (1)?
  a) 1110
  b) 1001
  c) 0111
  d) 1111

Signed vs. Unsigned in C

- char, short, int, and long
  - signed integer types
  - right shift (>>) is arithmetic
- unsigned char, unsigned short, unsigned int, unsigned long
  - unsigned integer types
  - right shift (>>) is logical

Why the signed integer types use the arithmetic right shift will be clear by the end of the next lecture.

Numeric Ranges

- **Unsigned Values**
  - *UMin* = 0
    000…0
  - *UMax* = $2^w - 1$
    111…1
- **Two's Complement Values**
  - *TMin* = -2^{w-1}
    100…0
  - *TMax* = $2^{w-1} - 1$
    011…1
- **Other Values**
  - Minus 1
    111…1

Values for *W* = 16

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

Supplied by CMU.

Values for Different Word Sizes

	W			
	8	**16**	**32**	**64**
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- **Observations**
  - |*TMin* | = *TMax* + 1
    - » Asymmetric range
  - *UMax* = 2 * *TMax* + 1
- **C Programming**
  - **#include** <limits.h>
  - declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - values platform-specific

Supplied by CMU.

Quiz 3

- What is –TMin (assuming two's complement signed integers)?
  a) TMin
  b) TMax
  c) 0
  d) 1

