C Program that Succeeds at Nothing

```c
int main() {
    return 0;
}
```
Compile and Run

% gcc x.c
% ./a.out

% gcc -o x x.c
% ./x
C Program that Fails at Nothing

```c
int main() {
    return 1;
}
```

a non-0 result reports failure
Enabling Warnings

% gcc -Wall -o x x.c
% ./x
C Program that Prints, But Makes gcc Complain

```c
int main() {
    printf("Hi\n");
    return 0;
}
```

Inside a string, \n means “newline” — and that’s true for C, Java, and most languages
C Program that Prints, And Keeps gcc Happy

#include <stdio.h>

int main() {
    printf("Hi\n");
    return 0;
}

#include is similar to import
C Program that Prints a Number

```c
#include <stdio.h>

int main() {
    printf("Ten and ten make %d\n", 10+10);
    return 0;
}
```

In a string passed to `printf`,

- `%d` means “print the next `int`”
- `%f` means “print the next `double`”
- `%s` means “print the next string”
- `%p` means “print the next address”
- `%c` means “print the next character”
Hexadecimal Numbers

```c
#include <stdio.h>

int main() {
    printf("Hex 10 and hex 10 make %d\n", 0x10 + 0x10);
    return 0;
}
```

0x starts a base-16 number
#include <stdio.h>

int main()
{
    printf("%p %p\n", main, printf);
    return 0;
}

Everything is a Number
Variables Live in Memory

```c
#include <stdio.h>

int main() {
    int v = 5;

    printf("At %p is %d\n", &v, v);
    return 0;
}
```

& as an operator means “the address of”
Variables Live in Memory

```c
#include <stdio.h>

int main()
{
    int v = 5;
    int* p = &v;

    v = 6;
    printf("At %p is %d\n", p, *p);
    return 0;
}
```

* in a type means “the address of a”

* as an operator means “value at the address”
Changing Memory can Change Variables

```c
#include <stdio.h>

int main() {
    int v = 5;
    int* p = &v;

    *p = 7;
    printf("V at end: %d\n", v);
    return 0;
}
```
#include <stdio.h>

int main() {
    int v = 5;
    int* p = &v;

    printf("At %p is %d\n", p, p[0]);
    return 0;
}

Array Notation Also Looks in an Address
Address Arithmetic

```c
#include <stdio.h>

int main() {
    int v = 5;
    int* p = &v;

    printf("At %p is %d\n", p+1, p[1]);
    return 0;
}
```

This particular result is unpredictable
#include <stdio.h>

int main() {
    int a[3] = { 1, 2, 3 };
    int* p = a;

    printf("%d, %d, %d\n",
            a[0], p[1], *(p + 2));
    return 0;
}

#include <stdio.h>

int main() {
    int a[3] = { 1, 2, 3 };
    int* p = a;
    int* q = &a;

    printf("%p = %p, but not %p\n", p, q, &p);
    return 0;
}

Special treatment of sized-array names makes [ ]-expression notation consistent
A String is an Array of Characters

```c
#include <stdio.h>

int main() {
    char* s = "apple";

    printf("%s: %c, %c, %c\n",
           s, s[0], s[1], *(s + 3));
    return 0;
}
```

Copy
Characters are Just Numbers

```c
#include <stdio.h>

int main() {
    char* s = "apple";
    printf("%s: %d, %d, %d\n",
           s, s[0], s[1], *(s + 3));
    return 0;
}
```
Arrays of Strings

#include <stdio.h>

int main() {
    char* s[3] = { "apple",
                   "banana",
                   "coconut" };

    char** ss = s;

    printf("%s (%c...), %s, %s\n",
           ss[0], ss[0][0], ss[1], s[2]);
    return 0;
}

Using Command-Line Arguments

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

int main(int argc, char** argv) {
    int a, b;

    a = atoi(argv[1]);
    b = atoi(argv[2]);

    printf("%d\n", a + b);

    return 0;
}
```

Don't use atoi() in real programs, because it doesn't distinguish good inputs from bad ones.
Sizes of Numbers

Each “box” in a machine’s memory holds a number between -128 and 127

... or 0 to 255, depending on how you look at it

• a char takes up one of them

• a short takes up two of them (-32768 to 32767)

• an int takes up four of them (-2147483648 to 2147483647)

• a long takes up four or eight, depending

• an address takes up four or eight, depending

  char*, int*, char**, etc.
#include <stdio.h>

int main() {
    char cs[2] = {0, 1};
    int  is[2] = {0, 1};

    printf("Goes up by 1: %p, %p\n", cs, cs+1);
    printf("Goes up by 4: %p, %p\n", is, is+1);

    return 0;
}
Computing Sizes

```c
#include <stdio.h>

int main()
{
    char cs[2] = {0, 1};

    printf("char size is %ld\n", sizeof(char));
    printf("char size is %ld\n", sizeof(cs[0]));
    printf("address size is %ld\n", sizeof(&cs));
    return 0;
}
```

The `sizeof` operator works on types or variables
#include <stdio.h>
#include <stdlib.h>

int main() {
    int* a;

    a = malloc(100 * sizeof(int));
a[99] = 5;
    printf("array at %p ends in %d\n", a, a[99]);

    return 0;
}
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    char *a;
    int len1, len2;

    len1 = strlen(argv[1]);
    len2 = strlen(argv[2]);
    a = malloc(len1 + len2 + 1);
    memcpy(a, argv[1], len1);
    memcpy(a + len1, argv[2], len2 + 1); // include terminator
    printf("%s\n", a);

    return 0;
}
More C: For Loops

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

int main(int argc, char** argv) {
    int i;
    int sum = 0;

    for (i = 1; i < argc; i++) {
        sum += atoi(argv[i]);
    }

    printf("%d\n", sum);

    return 0;
}
```

... just like Java
More C: Defining Functions

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

int twice(int n) {
    return n + n;
}

int main(int argc, char** argv) {
    printf("%d\n", twice(atoi(argv[1])));
    return 0;
}
```

... just like Java
Bytes and Bits

- Each address in memory contains a byte
- Each byte contains 8 bits

\[
\begin{align*}
0000 \ 0000 &= 0 \\
0000 \ 0001 &= 2^0 = 1 \\
0000 \ 0100 &= 2^2 = 4 \\
0010 \ 0100 &= 2^2 + 2^5 = 36 \\
1000 \ 0000 &= 2^7 = 128 \\
1111 \ 1111 &= 2^0 + \ldots + 2^7 = 255
\end{align*}
\]

This is the unsigned interpretation
Bytes and Bits

• Each address in memory contains a **byte**

• Each byte contains 8 **bits**

```
0000 0000  =  0
0000 0001  =  2^0 = 1
0000 0100  =  2^2 = 4
0010 0100  =  2^2 + 2^5 = 36
1000 0000  =  -2^7 = -128
1111 1111  =  2^0 + ... 2^6 - 2^7 = -1
```

This is the **signed** interpretation

a.k.a. **two's complement**
Multi-byte Encodings

To represent larger numbers, use multiple consecutive addresses, and refer to the first one

at 0x171

1111 1111

at 0x172

0000 0000

Unsigned with $w$ bits: \[ \sum_{i = 0}^{w-1} x_i 2^i \]

Signed with $w$ bits: \[ -x_{w-1} 2^{w-1} + \sum_{i = 0}^{w-2} x_i 2^i \]
Multi-byte Encodings

To represent larger numbers, use multiple consecutive addresses, and refer to the first one

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Bits</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 byte</td>
<td>8</td>
<td>-2(^7) to 2(^7)-1</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1 byte</td>
<td>8</td>
<td>0 to 2(^8)-1</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>16</td>
<td>-2(^{15}) to 2(^{15})-1</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
<td>32</td>
<td>-2(^{31}) to 2(^{31})-1</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4 bytes</td>
<td>32</td>
<td>0 to 2(^{32})-1</td>
</tr>
<tr>
<td>long</td>
<td>8 bytes</td>
<td>64</td>
<td>-2(^{63}) to 2(^{63})-1</td>
</tr>
</tbody>
</table>
Representing Information

The set of 32 bits

1111 0001 0100 0001 0100 0001 0100 0001

represents

an unsigned integer $> 2^{31}$
The set of 32 bits

\[ 1111 \ 0001 \ 0100 \ 0001 \ 0100 \ 0001 \ 0100 \ 0001 \]

represents

a negative integer
Logical Operations

C doesn’t distinguish booleans from numbers

• 0 counts as false
• any other value counts as true

Logical operations on numbers produce 0 or 1

• &&
• ||
• !

5 && 7 ⇒ 1  
5 && 0 ⇒ 0  
!(5 && 0) ⇒ 1
#include <stdio.h>

int main(int argc, char* argv[]) {
    int a = 0x0;
    int b = 0x93;

    printf("%x %x\n", !a, !b);

    printf("%x %x %x %x\n", a&&b, a||b, !a||!b, !a&&!b);

    return 0;
}
Bit Operations

C provides operations for manipulating bits within integers:

- & — bitwise AND
- | — bitwise OR
- ^ — bitwise XOR
- ~ — invert all bits

\[
\begin{align*}
5 \ & \ 7 & \Rightarrow & \ 5 \\
101 \ 111 \ 101 & \ & \ & \ 101 \\
5 \ & \ | \ 6 & \Rightarrow & \ 7 \\
101 \ 110 \ 111 & \ & \ & \ 0..11 \ 0..000 \ 1..111 \\
\sim (5 \ & \ \& \ 0) & \Rightarrow & \ -1 \\
0..11 \ 0..000 \ 1..111 & \ & \ & \ 101 \ 110 \ 011 \\
5 \ & \ ^\ \ 6 & \Rightarrow & \ 3 \\
101 \ 110 \ 011 & \ & \ & \ 101 \ 110 \ 011
\end{align*}
\]

Don’t confuse these with logical && and ||
#include <stdio.h>

int main(int argc, char* argv[]) {
    int a = 0x13; // 0001 0011
    int b = 0x55; // 0101 0101

    printf("%x %x %x\n", a|0, a&1, ~~a);

    printf("%x %x\n", a|a&b, a&(a|b));

    printf("%x %x\n", ~(a&b), ~a|~b);

    return 0;
}

Copy
Bit Shifting

Besides manipulating bits in place, operators can shift them around:

- \( \ll \) — shift bits left
- \( \gg \) — shift bits right, preserve sign

Bits that fall off the end are lost

\[
\begin{array}{llllll}
5 \ll 1 \Rightarrow 10 & 5 \gg 1 \Rightarrow 2 & 5 \gg 3 \Rightarrow 0 \\
101 & 1010 & 101 & 10 & 00101 & 00
\end{array}
\]
#include <stdio.h>

int main(void) {
    int a = 0x1;
    int b = 0x11;
    int c = 0x80000000;

    printf("%i %i %i\n", a<<1, a<<2, a<<3);
    printf("%i %i\n", b<<5, b*32);

    printf("%x %x %x %x\n", b, b>>1, b>>2, b>>3);
    printf("%x %x %x %x\n", c, c>>1, c>>2, c>>3);

    return 0;
}

Bit Shifting
Multiplication by Shifting and Adding

Shifting left by $N$ is the same as multiplying by $2^N$

To multiply $x$ by 3

- shift $x$ left by 1, ... which is the same as multiplying by 2
- then add $x$

To multiply $x$ by 10:

- shift $x$ left by 2, ... which is the same as multiplying by 4
- add $x$, ... brings us to 5
- then shift left again
#include <stdio.h>

int main()
{
    int x = 1000000000; // almost 2^30
    int y = 2000000000; // almost 2^31
    int z = x + y;
    printf("%d\n", z); // maybe -1294967296
    return 0;
}

Beware: the output of this program is undefined by the C standard
Non-overflow

```
#include <stdio.h>

int main()
{
    unsigned int x = 1000000000; // almost 2^30
    unsigned int y = 2000000000; // almost 2^31
    unsigned int z = x + y;     // less than 2^32
    printf("%u\n", z);         // definitely 3000000000
    return 0;
}
```

No unsigned int overflow
Non-overflow plus Coercion

#include <stdio.h>

int main()
{
    unsigned int x = 1000000000; // almost 2^30
    unsigned int y = 2000000000; // almost 2^31
    unsigned int z = x + y;   // less than 2^32
    printf("%d\n", z); // definitely -1294967296
    return 0;
}

No unsigned int overflow, and reinterpreting the unsigned int bits as int produces a specified result
Non-overflow plus Coercion

```c
#include <stdio.h>

int main()
{
    unsigned int x = 1000000000; // almost 2^30
    unsigned int y = 2000000000; // almost 2^31
    int z = x + y;
    printf("%d\n", z); // definitely -1294967296
    return 0;
}
```

Earlier coercion has the same effect, as long as its after arithmetic
Integer Conversions

signed $\leftrightarrow$ unsigned at same size
  keep the same bits

smaller $\Rightarrow$ larger
  - unsigned  
    pad with zeroes
  - signed
    pad with sign bit

larger $\Rightarrow$ smaller
  - unsigned
    drop extra bits
  - signed
    drop extra bits, then copy sign bit

  e.g., int $\leftrightarrow$ unsigned
  e.g., unsigned char $\Rightarrow$ unsigned
  e.g., char $\Rightarrow$ int
  e.g., unsigned long $\Rightarrow$ unsigned
  e.g., long $\Rightarrow$ int
#include <stdio.h>

int main()
{
    char c = -5;
    int i = c;
    unsigned u = i;
    unsigned u2 = c;

    printf("%d %u %u\n", i, u, u2);
    return 0;
}
Non-Integer Numbers

The float and double types implement floating-point numbers of the form

\[ \pm M \times 2^{E} \]

float  ±: 1 bit  M: 23 bits  ±E: 8 bits  = 32 bits

double ±: 1 bit  M: 52 bits  ±E: 11 bits  = 64 bits

Constraints on M and ±E make good use of the bits
Normalized: $\pm E$ is not its maximum or minimum value

$$1 \leq M < 2$$

$$\pm E = e + 1 - 2^{k-1}$$

$$M = 1 + f/2^n$$

Denormalized: $\pm E$ is its minimum value (which is negative)

$$0 \leq M < 1$$

$$\pm E = 2 - 2^{k-1}$$

$$M = f/2^n$$

Infinity: $\pm E$ is its maximum value

$$\pm 2^{k-1}$$

$$0$$

Not-a-Number: $\pm E$ is its maximum value (many representations!)

$$\pm 2^{k-1}$$

$$\text{non-0}$$
Floating-Point Decoding

\[ k = 8 \quad n = 23 \]

0010010001000000000000000000000000

0 \Rightarrow \text{positive} \quad 01001000 \Rightarrow \text{normalized}

1 \leq M < 2

\[ \pm 0 < \pm E + 2^{k-1} - 1 < 2^{k-1} \]

\[ \pm E = e + 1 - 2^{k-1} \]

\[ M = 1 + f/2^n \]

01001000 \Rightarrow e = 72

\Rightarrow E = 72 + 1 - 128 = -55

\Rightarrow M = 1 + 2^{22}/2^{23} = 1.5

\Rightarrow 1.5 \times 2^{-55} \approx 4.16334 \times 10^{-17}
Floating-Point Decoding

\[ k = 8 \quad \quad \quad \quad n = 23 \]

\[
\begin{array}{cccccccccccccccccccc}
0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccc}
0x2 & 0x4 & 0x4 & 0x0 & 0x0 & 0x0 & 0x0 & 0x0
\end{array}
\]

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

int main() {
    int i = 0x24400000;
    float f;
    memcpy(&f, &i, sizeof(float));
    printf("%g\n", f);
    return 0;
}

\[ \Rightarrow 1.5 \times 2^{-55} \approx 4.16334 \times 10^{-17} \]
Floating-Point Decoding

\[ k = 8 \quad n = 23 \]

1 \quad \Rightarrow \text{negative}

00000000 \quad \Rightarrow \text{denormalized}

\[ 0 \leq M < 1 \]

\[ \pm E = 2 - 2^{k-1} \]

\[ M = f/2^n \]

\[ 00000000 \]

\[ \Rightarrow \quad E = 2 - 128 = -126 \]

100000000000000000000000000000

\[ \Rightarrow \quad f = 2^{22} \]

\[ \Rightarrow \quad M = 2^{22}/2^{23} = 0.5 \]

\[ \Rightarrow \quad 0.5 \times 2^{-126} \approx 5.87747 \times 10^{-39} \]
Floating-Point Decoding

\[ k = 8 \quad \text{and} \quad n = 23 \]

\[
\begin{array}{cccccccccccccccccccc}
  1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccc}
  0 & x & 8 & \quad 0 & x & 0 & \quad 0 & x & 4 & \quad 0 & x & 0 & \quad 0 & x & 0 & \quad 0 & x & 0 & \quad 0 & x & 0 & \quad 0 & x & 0
\end{array}
\]

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

int main() {
    int i = 0x80400000;
    float f;
    memcpy(&f, &i, sizeof(float));
    printf("\%g\n", f);
    return 0;
}

⇒ \[0.5 \times 2^{-126} \approx 5.87747 \times 10^{-39}\]
Floating-Point Decoding

\[ k = 8 \quad n = 23 \]

011111111000000000000000000000000

0 ⇒ positive \quad 11111111 ⇒ special

**Infinity:** ±\(E\) is its maximum value

\[ \begin{array}{c|c|c}
± & 2^k - 1 & 0 \\
\hline
\end{array} \]

**Not-a-Number:** ±\(E\) is its maximum value (many representations!)

\[ \begin{array}{c|c|c}
± & 2^k - 1 & \text{non-0} \\
\hline
\end{array} \]

00000000000000000000000000000000 ⇒ infinity
Floating-Point Decoding

\[ k = 8 \quad \text{and} \quad n = 23 \]

\[ 01111111110000000001100000000000000 \]

\[ 0 \Rightarrow \ldots \quad 11111111 \Rightarrow \text{special} \]

\textbf{Infinity:} \( \pm E \) is its maximum value

\[ \pm \begin{array}{c} 2^{k-1} \end{array} \quad \text{non-0} \]

\textbf{Not-a-Number:} \( \pm E \) is its maximum value \hspace{1cm} \text{(many representations!)}

\[ \pm \begin{array}{c} 2^{k-1} \end{array} \quad \text{not-a-number} \]

\[ 00000000011000000000000000 \Rightarrow \text{not-a-number} \]
Representing Information

The set of 32 bits

1111 0001 0100 0001 0100 0001 0100 0001

represents

an unsigned integer > $2^{31}$
Representing Information

The set of 32 bits

1111 0001 0100 0001 0100 0001 0100 0001

represents

a negative integer
Representing Information

The set of 32 bits

1111 0001 0100 0001 0100 0001 0100 0001

represents

a floating-point value close to \(-9.57 \times 10^{29}\)
Multi-byte Ordering

When we use an address for a multi-byte encoding, which end is it?

- **Big endian** — that’s the number
  
  \[1111\ 1111\ 0000\ 0000 = 65280\]

- **Little endian** — that’s the number
  
  \[0000\ 0000\ 1111\ 1111 = 255\]

\[x86-64\ is\ little\ endian\]
Representing Information

The set of 32 bits

<table>
<thead>
<tr>
<th>Address</th>
<th>Binary Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xa051</td>
<td>0100 0001</td>
</tr>
<tr>
<td>0xa052</td>
<td>0100 0001</td>
</tr>
<tr>
<td>0xa053</td>
<td>0100 0001</td>
</tr>
<tr>
<td>0xa054</td>
<td>1111 0001</td>
</tr>
</tbody>
</table>

represents

an unsigned integer > $2^{31}$
Representing Information

The set of 32 bits

<table>
<thead>
<tr>
<th>at 0xa051</th>
<th>at 0xa052</th>
<th>at 0xa053</th>
<th>at 0xa054</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100 0001</td>
<td>0100 0001</td>
<td>0100 0001</td>
<td>1111 0001</td>
</tr>
</tbody>
</table>

represents

a negative integer
Representing Information

The set of 32 bits

\[
\begin{array}{cccc}
\text{at 0xa051} & \text{at 0xa052} & \text{at 0xa053} & \text{at 0xa054} \\
0100 0001 & 0100 0001 & 0100 0001 & 1111 0001 \\
\end{array}
\]

represents

a floating-point value close to $-9.57 \times 10^{29}$
Characters and Strings

A letter like “a” is represented as a byte

```
char = 1 byte
```

**ASCII** encoding:

```
'a' = 97
'b' = 98
...  
'A' = 65
'B' = 66
...  
'0' = 48
'1' = 49
...  
```
Characters and Strings

A letter like “a” is represented as a byte

\[ \texttt{char} = 1 \text{ byte} \]

A string like “apple” is represented as a sequence of bytes terminated with a 0 byte

<table>
<thead>
<tr>
<th>at 0xf01</th>
<th>at 0xf02</th>
<th>at 0xf03</th>
<th>at 0xf04</th>
<th>at 0xf05</th>
<th>at 0xf06</th>
</tr>
</thead>
<tbody>
<tr>
<td>97</td>
<td>112</td>
<td>112</td>
<td>108</td>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>'a'</td>
<td>'p'</td>
<td>'p'</td>
<td>'l'</td>
<td>'e'</td>
<td></td>
</tr>
</tbody>
</table>

\[ \texttt{char*} = 8 \text{ bytes} = \text{the address of many bytes} \]
#include <stdio.h>

int main() {
    char* s = "apple";

    printf("%s: %d, %d, %d\n",
           s, s[0], s[1], s[2]);
    return 0;
}

Characters are Just Numbers
Representing Information

The set of 32 bits

<table>
<thead>
<tr>
<th>at 0xa051</th>
<th>at 0xa052</th>
<th>at 0xa053</th>
<th>at 0xa054</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100 0001</td>
<td>0100 0001</td>
<td>0100 0001</td>
<td>1111 0001</td>
</tr>
</tbody>
</table>

represents

an unsigned integer $> 2^{31}$
Representing Information

The set of 32 bits

\[
\begin{array}{cccc}
\text{at } 0xa051 & \text{at } 0xa052 & \text{at } 0xa053 & \text{at } 0xa054 \\
0100 \ 0001 & 0100 \ 0001 & 0100 \ 0001 & 1111 \ 0001 \\
\end{array}
\]

represents

a negative integer
Representing Information

The set of 32 bits

\[
\begin{array}{cccc}
\text{at } 0xa051 & \text{at } 0xa052 & \text{at } 0xa053 & \text{at } 0xa054 \\
0100 0001 & 0100 0001 & 0100 0001 & 1111 0001 \\
\end{array}
\]

represents

a floating-point value close to \(-9.57 \times 10^{29}\)
Representing Information

The set of 32 bits

\[
\begin{array}{cccc}
\text{at 0xa051} & \text{at 0xa052} & \text{at 0xa053} & \text{at 0xa054} \\
0100 0001 & 0100 0001 & 0100 0001 & 1111 0001
\end{array}
\]

represents

four characters: A A A ñ
Representing Information

The set of 32 bits

\[
\begin{array}{cccc}
\text{at 0xa051} & \text{at 0xa052} & \text{at 0xa053} & \text{at 0xa054} \\
0100 0001 & 0100 0001 & 0100 0001 & 1111 0001 \\
\end{array}
\]

represents

machine instructions to try to execute
```c
#include <stdio.h>

int main() {
    char* s = "apple";
    short* p = (short*)s;

    printf("%s: %d %d\n", s, *p, s[0] + (s[1] * 256));
    return 0;
}
```
Casts

```c
#include <stdio.h>

int main() {
    float f = 2.5;
    int i = *(int *)&f;
    printf("%f %d\n", f, i);
    return 0;
}
```

This kind of cast is generally undefined in standard C
“Casts” via memcpy

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

int main() {
    float f = 2.5;
    int i;
    memcpy(&i, &f, sizeof(int));

    printf("%f %d\n", f, i);

    return 0;
}
```

The result is defined for a little-endian
C Practicalities

- “word” refers to `sizeof(int*)` bytes
  - e.g., 64-bit or 32-bit word sizes
  - ... except when it doesn’t

- `int` not necessarily two’s complement, by standard
  - always two’s complement in practice

- sizes of `int`, `short`, `long` not specified by standard
  - `int` is 4 bytes in practice
  - `long` can be 4 or 8 bytes
  - `intptr_t` matches an address size

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

int main(void) {
    int i;
    float f = 0.0;

    for (i = 0; i < 10; i++) {
        f = f + 0.1;
    }

    printf("%f\n", f);
    return 0;
}

\textbf{f} is not 1.0, but too few digits shown by default
Limits for Signed Integers

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

int check_grow (int x) {
    return (x+1) > x;
}

int main (void) {
    printf (%d\n%, (INT_MAX+1) > INT_MAX);
    printf (%d\n%, check_grow(INT_MAX));
    return 0;
}
```

Result depends on optimization level, `-O2` or not, which is a sign of a broken program
Limits for Unsigned Integers

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

int check_grow (unsigned x) {
    return (x+1) > x;
}

int main (void) {
    printf ("%d\n", (UINT_MAX+1) > UINT_MAX);
    printf ("%d\n", check_grow(UINT_MAX));
    return 0;
}
```

Result is well defined
# Casts and Aliasing

```
#include <stdio.h>

void set(int *i, float *f, int *j) {
    printf("f at %p, j at %p\n", f, j);
    *j = 1;
    *f = 0.0; /* what if `j` and `f` at same address? */
    *i = *j;
}

int main (void) {
    int i, j;
    set(&i, (float *)&j, &j);
    printf ("%d %d\n", i, j);
    return 0;
}
```

Result depends on optimization level...