C Program that Succeeds at Nothing

```c
int main() {
  return 0;
}
```

Copy

Compile and Run

```
% gcc x.c
% ./a.out
```

```
% gcc -o x x.c
% ./x
```

C Program that Fails at Nothing

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

Copy

a non-0 result reports failure

Enabling Warnings

```
% gcc -Wall -o x x.c
% ./x
```

C Program that Prints, But Makes gcc Complain

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

[Copy]

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;
}
```

Copy

#include is similar to **import**

C Program that Prints a Number

```
#include <stdio.h>

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

Copy

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

```
#include <stdio.h>

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

Copy

0x starts a base-16 number

Everything is a Number

```
#include <stdio.h>

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

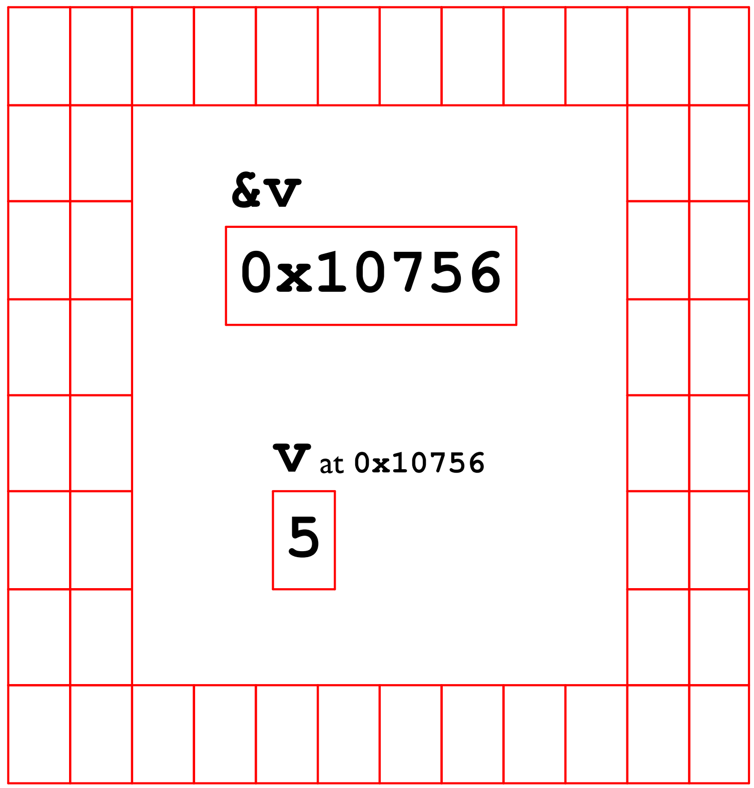
Copy

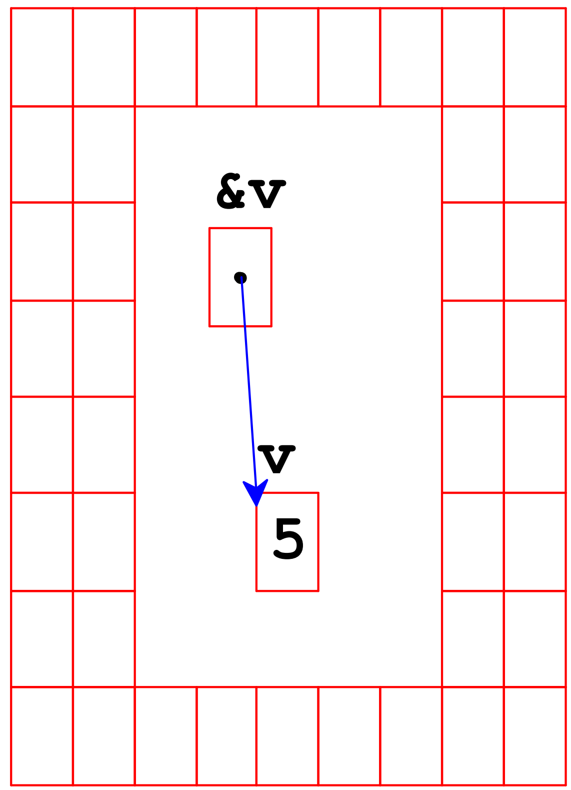
Variables Live in Memory

```
#include <stdio.h>

int main() {
  int v = 5;

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

Copy

& as an operator means “the address of”


&v
0x10756
v at 0x10756
5


&v
v
5

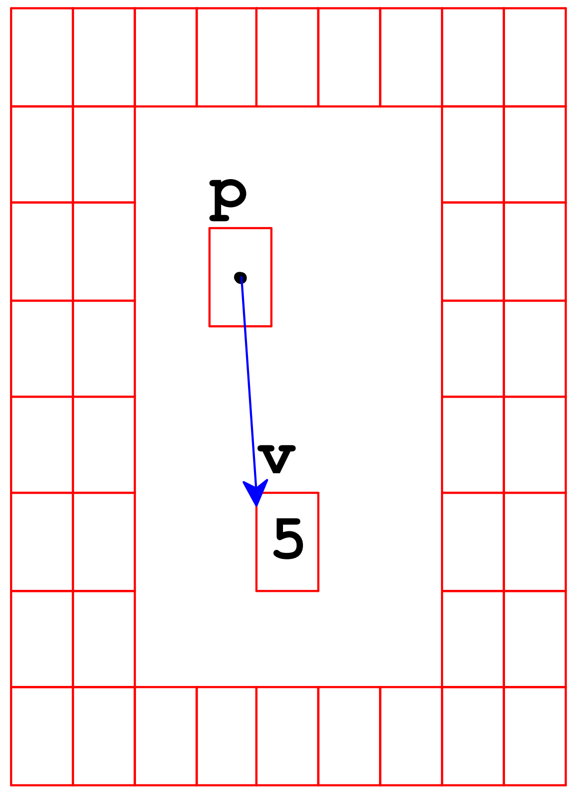
Variables Live in Memory

```
#include <stdio.h>

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

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

[Copy]

* in a type means "the address of a"

* as an operator means "value at the address"


p
v
5

Changing Memory can Change Variables

```
#include <stdio.h>

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

  *p = 7;
  printf("V at end: %d\n", v);
  return 0;
}
```

Copy

Array Notation Also Looks in an Address

```
#include <stdio.h>

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

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

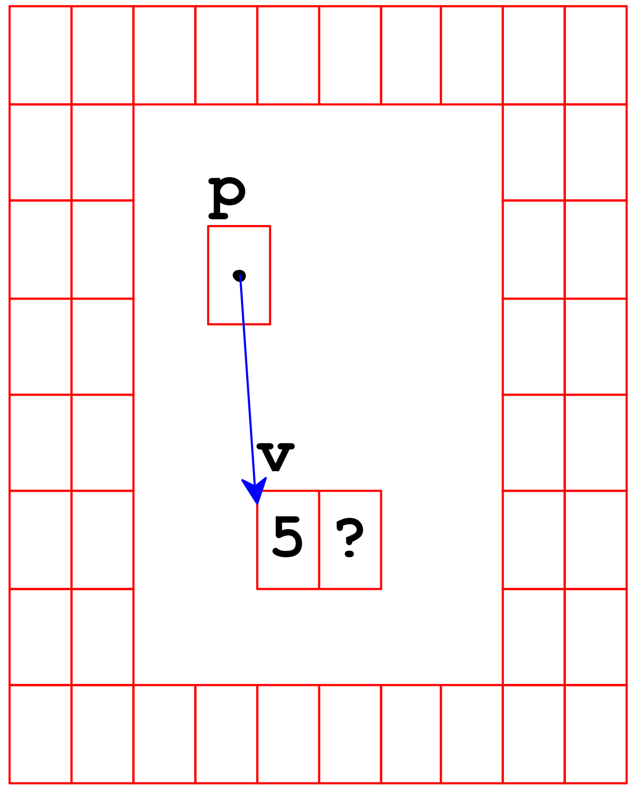
Copy

Address Arithmetic

```
#include <stdio.h>

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

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

Copy

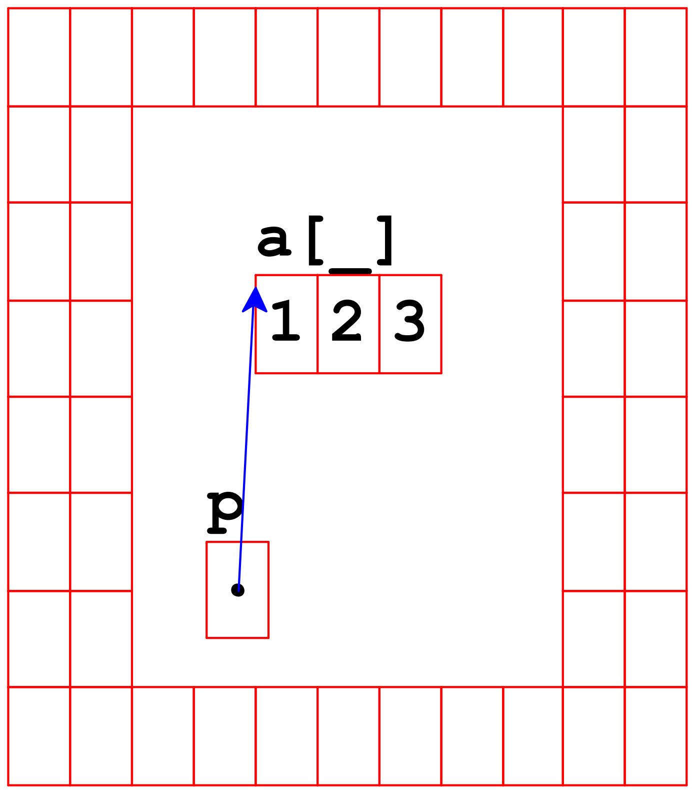
This particular result is unpredictable


p
v
5
?

Arrays Take up Memory

```
#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;
}
```

Copy

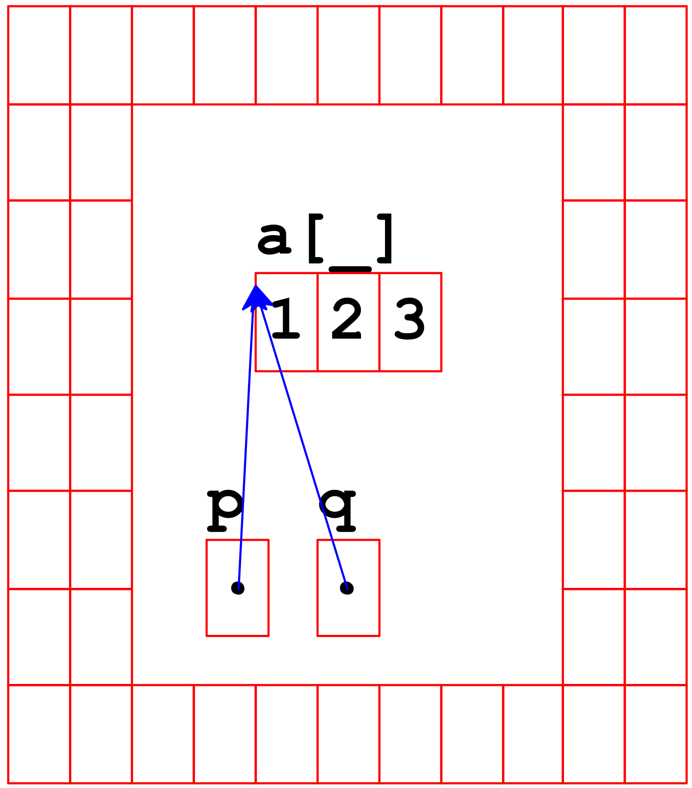

a[_]
1
2
3
p

Array Names Are a Little Strange

```
#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;
}
```

Copy

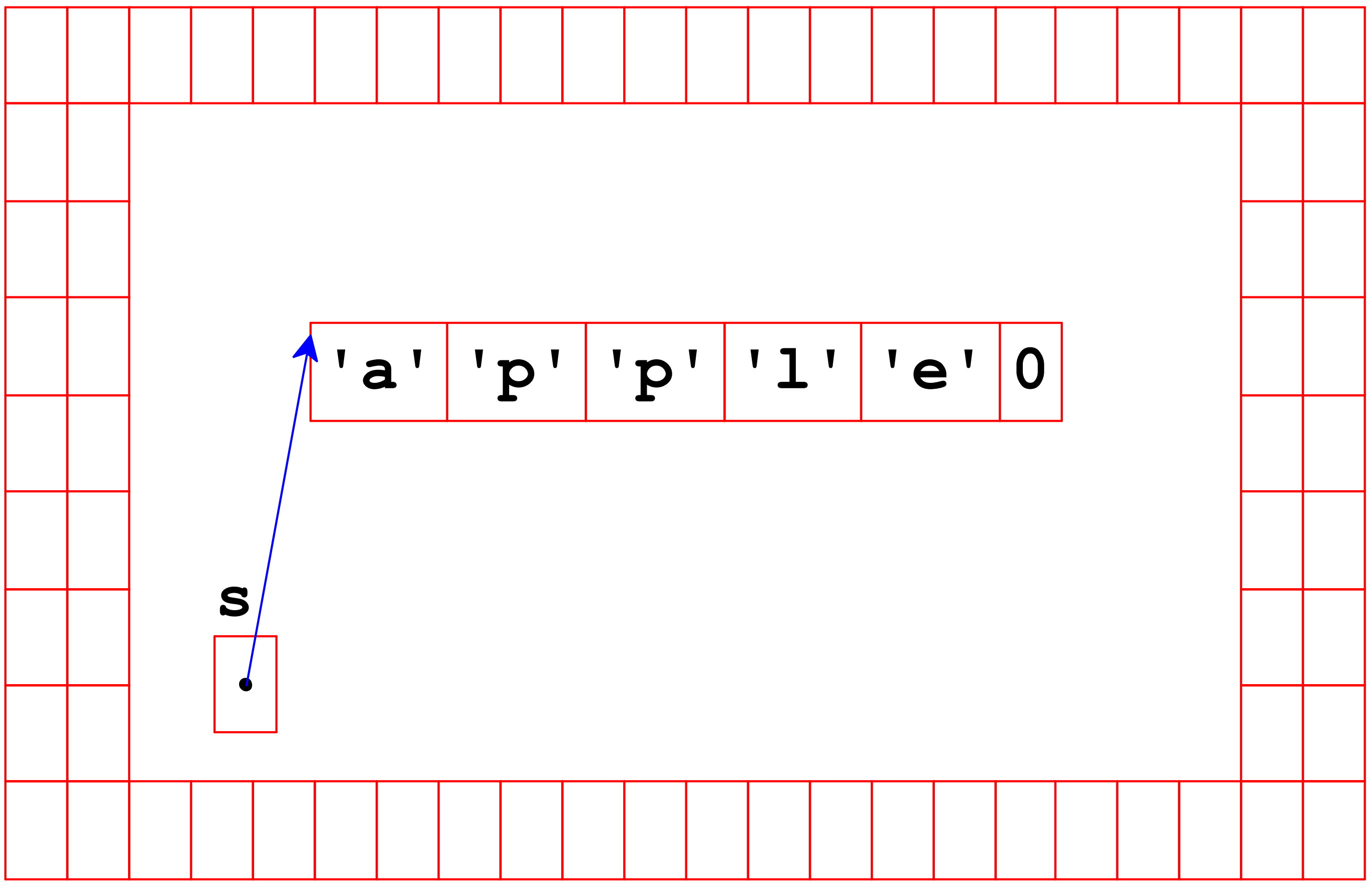
Special treatment of sized-array names makes `[]`-expression notation consistent

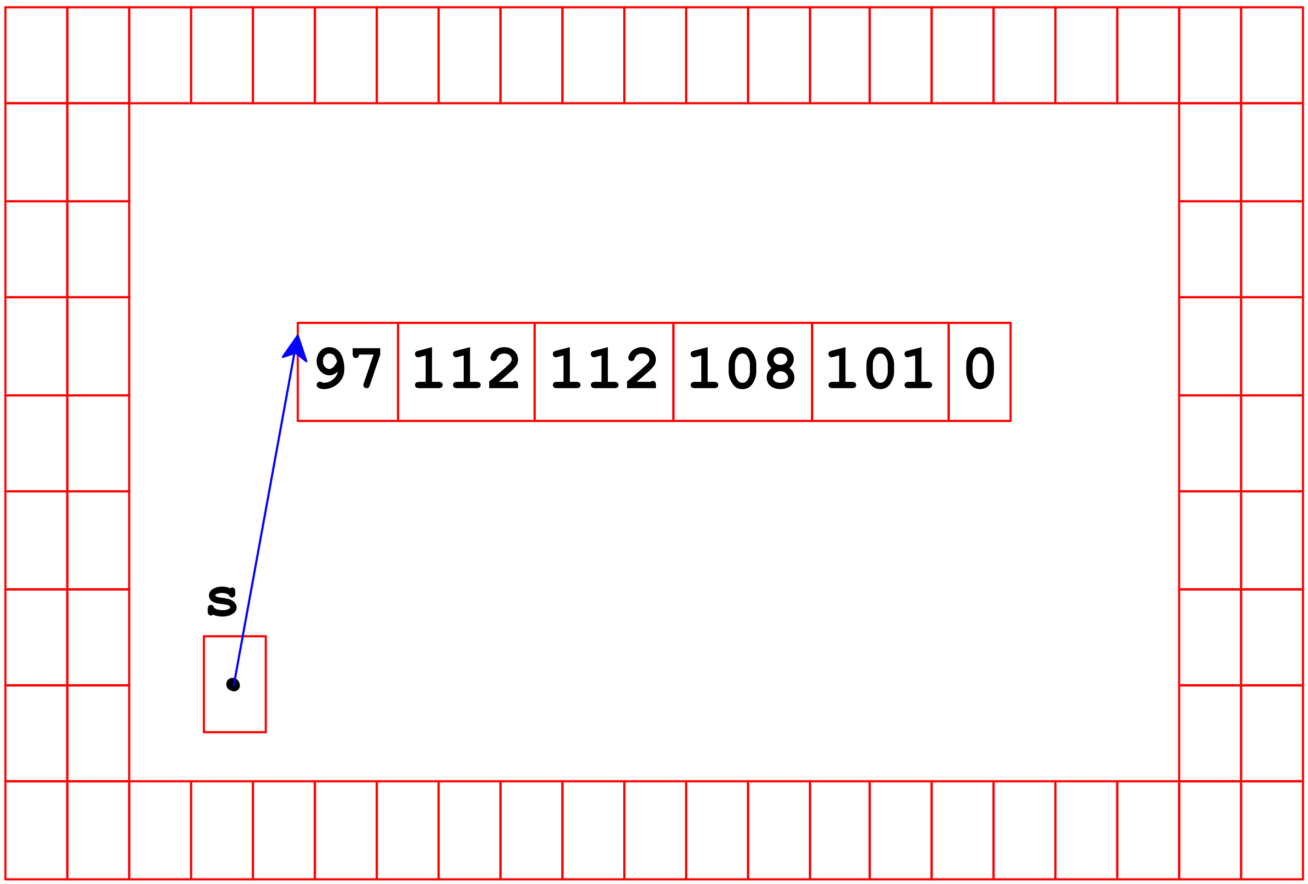

a[_]
1
2
3
p
q

A String is an Array of Characters

```
#include <stdio.h>

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

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

Copy


'a' 'p' 'p' 'l' 'e' 0
s


97
112
112
108
101
0
s

Characters are Just Numbers

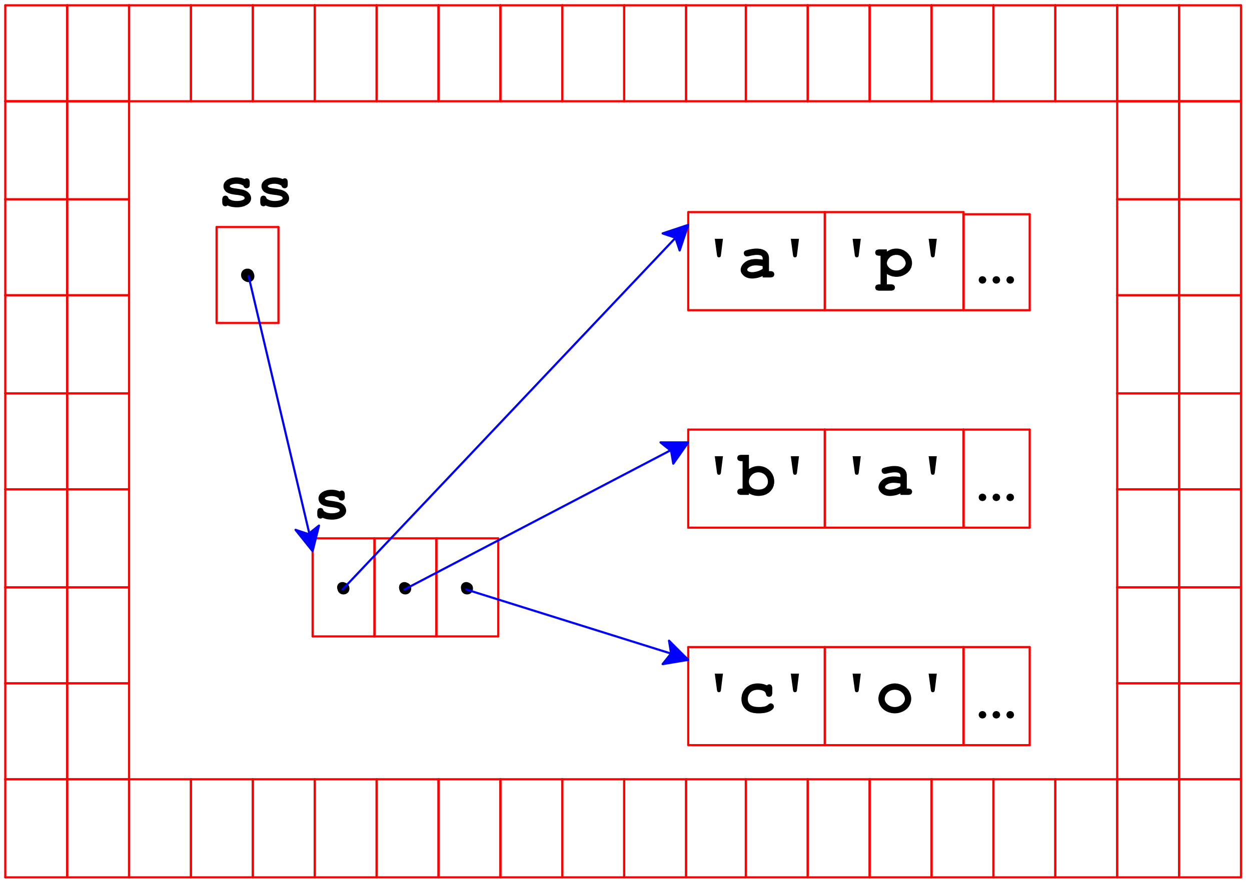
```
#include <stdio.h>

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

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

Copy

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;
}
```

Copy


ss
s
'a' 'p' ...
'b' 'a' ...
'c' 'o' ...

Using Command-Line Arguments

```
#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;
}
```

Copy

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.

Pointer Arithmetic

```
#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;
}
```

Copy

Computing Sizes

```
#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;
}
```

Copy

The **`sizeof`** operator works on types or variables

Allocation

```c
#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;
}
```

Copy

Concatenation

```
#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;
}
```

Copy

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;
}
```

Copy

... just like Java

More C: Defining Functions

```
#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;
}
```

Copy

... just like Java

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	=	$2^0 +\dots 2^7 = 255$

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` = $2^0 + \ldots 2^6 - 2^7 = -1$

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`	at `0x172`
`1111 1111`	`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

at `0x171`	at `0x172`
`1111 1111`	`0000 0000`

`char`	1 byte	8 bits	-2^{7} to $2^{7}-1$
`unsigned char`	1 byte	8 bits	0 to $2^{8}-1$
`short`	2 bytes	16 bits	-2^{15} to $2^{15}-1$
`int`	4 bytes	32 bits	-2^{31} to $2^{31}-1$
`unsigned int`	4 bytes	32 bits	0 to $2^{32}-1$
`long`	8 bytes	64 bits	-2^{63} to $2^{63}-1$

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

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`

Logical Operations

```
#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;
}
```

Copy

Bit Operations

C provides operations for manipulaing bits within integers:

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

`5 & 7` ⇒ `5`	`5	6` ⇒ `7`	`~(5 & 0)` ⇒ `-1`
`101 111 101`	`101 110 111`	`0..101 0..000 1..111`

`5 ^ 6` ⇒ `3`

`101 110 011`

Don't confuse these with logical `&&` and `||`

Bit Operations

```
#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:

- **<<** — shift bits left
- **>>** — shift bits right, preserve sign

Bits that fall off the end are lost

`5 << 1` $\Rightarrow$ **`10`**	**`5 >> 1`** $\Rightarrow$ **`2`**	**`5 >> 3`** $\Rightarrow$ **`0`**
`101` `1010`	`101` `10`	`00101` `00`

Bit Shifting

```
#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;
}
```

Copy

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

Overflow

```
#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;
}
```

Copy

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;
 }
```

Copy

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;
 }
```

Copy

No **`unsigned int`** overflow, and reinterpreting the **`unsigned int`** bits as **`int`** produces a specified result

Non-overflow plus Coercion

```
#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;
}
```

Copy

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

Integer Conversions

signed ⇔ unsigned at same size — e.g., **`int`** ⇔ **`unsigned`**

keep the same bits

smaller ⇒ larger

- unsigned — e.g., **`unsigned char`** ⇒ **`unsigned`**

  pad with zeroes

- signed — e.g., **`char`** ⇒ **`int`**

  pad with sign bit

larger ⇒ smaller

- unsigned — e.g., **`unsigned long`** ⇒ **`unsigned`**

  drop extra bits

- signed — e.g., **`long`** ⇒ **`int`**

  same value if fits, unspecified otherwise

Integer Conversions

```
#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;
}
```

Copy

Non-Integer Numbers

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

$$\pm M \times 2^{\pm E}$$

`float`	$\pm$: 1 bit	M: 23 bits	$\pm E$: 8 bits	= 32 bits
`double`	$\pm$: 1 bit	M: 52 bits	$\pm E$: 11 bits	= 64 bits

Constraints on M and $\pm E$ make good use of the bits

$\boxed{\pm M \times 2^{\pm E}}$ 1 bit for $\pm$ $\quad$ k bits for $\pm E$ $\quad$ n bits for M

$k = 8$ or 11 $\quad$ $n = 23$ or 52

Normalized: $\pm E$ is not its maximum or minimum value

$1 \leq M < 2$

$\pm$	$0 < \pm E + 2^{k-1} - 1 < 2^k - 1$	$(M-1)2^n$

$\pm E = e + 1 - 2^{k-1}$ $\quad$ $M = 1 + f/2^n$

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

$0 \leq M < 1$

$\pm$	0	$M2^n$

$\pm E = 2 - 2^{k-1}$ $\quad$ $M = f/2^n$

Infinity: $\pm E$ is its maximum value

$\pm$	$2^k - 1$	0

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

$\pm$	$2^k - 1$	non-0

Floating-Point Decoding

$k = 8$ $\quad$ $n = 23$

0	01001000	10000000000000000000000

0 ⇒ positive $\qquad$ 01001000 ⇒ **normalized**

$1 \le M < 2$

±	$0 < \pm E + 2^{k-1} - 1 < 2^k - 1$	$(M-1)2^n$

$\pm E = e + 1 - 2^{k-1}$ $\qquad$ $M = 1 + f/2^n$

01001000 $\Rightarrow$ $e = 72$

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

10000000000000000000000 $\Rightarrow$ $f = 2^{22}$

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

$\Rightarrow$ $1.5\times 2^{-55} \approx 4.16334\times 10^{-17}$

Floating-Point Decoding

$k = 8$ $\qquad\qquad$ $n = 23$

0010	0100	0100	0000	0000	0000	0000	0000
0x2	0x4	0x4	0x0	0x0	0x0	0x0	0x0

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

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

Copy

$\Rightarrow \quad 1.5\times2^{-55} \approx 4.16334\times10^{-17}$

Floating-Point Decoding

$k = 8$ $n = 23$

1	00000000	10000000000000000000000

1 ⇒ negative   00000000 ⇒ **denormalized**

$0 \leq M < 1$

±	0	$M2^n$
	$\pm E = 2\text{-}2^{k\text{-}1}$	$M = f/2^n$

00000000

$\Rightarrow \quad E = 2\text{-}128 = \text{-}126$

10000000000000000000000 $\Rightarrow \quad f = 2^{22}$

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

$\Rightarrow \quad 0.5\times 2^{-126} \approx 5.87747\times 10^{-39}$

Floating-Point Decoding

$k = 8$ $n = 23$

<table>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td colspan="4">0x8</td><td colspan="4">0x0</td><td colspan="4">0x4</td><td colspan="4">0x0</td><td colspan="4">0x0</td><td colspan="4">0x0</td><td colspan="4">0x0</td><td colspan="4">0x0</td></tr>
</table>

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

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

Copy

$\Rightarrow \quad 0.5\times2^{-126} \approx 5.87747\times10^{-39}$

Floating-Point Decoding

$k = 8$ $n = 23$

0	11111111	00000000000000000000000

0 ⇒ positive 11111111 ⇒ **special**

Infinity: $\pm E$ is its maximum value

±	2^k-1	0

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

±	2^k-1	non-0

00000000000000000000000 ⇒ infinity

Floating-Point Decoding

$k = 8$ $n = 23$

0 | 11111111 | 00000000011000000000000

0 ⇒ …   11111111 ⇒ **special**

Infinity: $\pm E$ is its maximum value

±	2^k-1	0

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

±	2^k-1	non-0

00000000011000000000000 ⇒ 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 -9.57×10^{29}

Multi-byte Ordering

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

at `0x171`	at `0x172`
`1111 1111`	`0000 0000`

- ***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

at `0xa051`	at `0xa052`	at `0xa053`	at `0xa054`
`0100 0001`	`0100 0001`	`0100 0001`	`1111 0001`

represents

an unsigned integer > 2^{31}

Representing Information

The set of 32 bits

at `0xa051`	at `0xa052`	at `0xa053`	at `0xa054`
`0100 0001`	`0100 0001`	`0100 0001`	`1111 0001`

represents

a negative integer

Representing Information

The set of 32 bits

at `0xa051`	at `0xa052`	at `0xa053`	at `0xa054`
`0100 0001`	`0100 0001`	`0100 0001`	`1111 0001`

represents

a floating-point value close to -9.57×10^{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

`char` = 1 byte

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

at **`0xf01`**	at **`0xf02`**	at **`0xf03`**	at **`0xf04`**	at **`0xf05`**	at **`0xf06`**
`97`	**`112`**	**`112`**	**`108`**	**`101`**	**`0`**
`'a'`	**`'p'`**	**`'p'`**	**`'l'`**	**`'e'`**	

`char*` = 8 bytes = the address of many bytes

Characters are Just Numbers

```
#include <stdio.h>

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

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

Copy

Representing Information

The set of 32 bits

at `0xa051`	at `0xa052`	at `0xa053`	at `0xa054`
`0100 0001`	`0100 0001`	`0100 0001`	`1111 0001`

represents

an unsigned integer > 2^{31}

Representing Information

The set of 32 bits

at `0xa051`	at `0xa052`	at `0xa053`	at `0xa054`
`0100 0001`	`0100 0001`	`0100 0001`	`1111 0001`

represents

a negative integer

Representing Information

The set of 32 bits

at `0xa051`	at `0xa052`	at `0xa053`	at `0xa054`
`0100 0001`	`0100 0001`	`0100 0001`	`1111 0001`

represents

a floating-point value close to -9.57×10^{29}

Representing Information

The set of 32 bits

at `0xa051`	at `0xa052`	at `0xa053`	at `0xa054`
0100 0001	0100 0001	0100 0001	1111 0001

represents

four characters: A A A ñ

Representing Information

The set of 32 bits

at `0xa051`	at `0xa052`	at `0xa053`	at `0xa054`
`0100 0001`	`0100 0001`	`0100 0001`	`1111 0001`

represents

machine instructions to try to execute

Casts

```
#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;
}
```

Copy

Casts

```c
#include <stdio.h>

int main() {
  float f = 2.5;
  int i = *(int *)&f;

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

  return 0;
}
```

Copy

This kind of cast is generally undefined in standard C

“Casts” via `memcpy`

```
#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;
}
```

Copy

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

`#include <inttypes.h>`

Adding Floating-Point Numbers

```
#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;
}
```

Copy

`f` is not `1.0`, but too few digits shown by default

Limits for Signed Integers

```
#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;
}
```

Copy

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

Limits for Unsigned Integers

```
#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;
}
```

Copy

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;
}
```

Copy

Result depends on optimization level...