DPST1092 24T3 — Data Representation and Integers

https://www.cse.unsw.edu.au/~dp1092/24T3/

Revision (1)

What happens when we declare variables in C?

What is an integer? How is it stored?

What happens when we malloc some memory? What do we get back?

What are pointers? What do they store?

Revision (2)

What happens when we declare variables in C?

''' c int x;

int y = 7; '''

When these lines are executed, what does it look like in the computer?

Revision (3) Names

What happens when we declare variables in C?

''' c int x;

x = 7; '''

When these lines are executed, what does it look like in the computer?

A chunk of memory from our RAM (more about this later) gets allocated to our program, and is *named* x.

And then the number 7 is written into the memory *named* x

Revision (4) Types

What happens when we declare variables in C?

''' c int x;

x = 7;

//What if we try and set it to the wrong type?

long long y = 99; //(long long is just another datatype that stores REALLY big numbers)

x = y; '''

Revision (5) Types cont.

What happens when we declare variables in C?

''' c int x;

x = 7;

What if we try and set it to the wrong type?

long long y = 99; //(long long is just another datatype that stores REALLY big numbers)

x = y;

That's right! It says you can't. Different compilers will tell you different things but usually something like:

'warning: conversion from 'long long int' to 'int' may change value [-Wconversion]'

Why did it gives us this warning?

Revision (6) Types cont.

What happens when we declare variables in C?

''' c int x;

x = 7;

//What if we try and set it to the wrong type?

//long long y = 99; //(long long is just another datatype that stores REALLY big numbers)

x = y; '''

That's right! It says you can't. Different compilers will tell you different things but usually something like:

'warning: conversion from 'long long int' to 'int' may change value [-Wconversion]'

Why did it gives us this warning?

It might not fit in the original integer.

Revision (7) Location

What happens when we declare variables in C?

int x;

x = 7;

Now how do we actually use this memory? Suppose we want to print our value. Which of the following should we run?

a: printf("%d", x);

b: printf("%d", &x);

Revision (8) Location cont.

What happens when we declare variables in C?

int x;

x = 7;

Now how do we actually use this memory? Suppose we want to print our value. Which of the following should we run?

a: printf("%d", x); (Wrong)

b: printf("%d", &x); (Correct)

Why is the first one correct? Let's look at the man pages.

Revision (9) Location cont.

What happens when we declare variables in C?

int x;

x = 7;

Now how do we actually use this memory? Suppose we want to print our value. Which of the following should we run?

a: printf("%d", x); (Wrong)

b: printf("%d", &x); (Correct)

Why is the first one correct? Printf requires the memory address or 'location' of what you want to print.

That is what the '&' does. You can think of it like a function, it takes in a name 'x' and then gives you the 'location' of x.

So &x will give you the location of x.

Printf then goes and prints the 'value' at that location.

Revision (10) Declaration Recap

What happens when we declare variables in C?

int x = 7;

So when we declare a variable in C, we are given a chunk of memory that has:

- a *name*, in this case 'x'.
- a *value*, in this case, 7.
- a *location*, retrievable via '&x'
- a *type*, which tells us...
  - its *size* (you can get this in bytes using 'sizeof')
  - what we can do with it (e.g. z = x + y works with integers since c knows + is trying to add them)

Revision (11) Pointers & Malloc

What happens when we declare variables in C?

int x = 7;

So when we declare a variable in C, we are given a chunk of memory that has:

- a *name*, in this case 'x'.
- a *value*, in this case, 7.
- a *location*, retrievable via '&x'
- a *type*, which tells us...
  - its *size* (you can get this in bytes using 'sizeof')
  - what we can do with it (e.g. z = x + y works with integers since c knows + is trying to add them)

Though not all the time will it have all these fields. Sometimes, there is no *name*. When might a variable not have a name?

Revision (12) Pointers & Malloc cont.

What happens when we declare variables in C?

int x = 7;

So when we declare a variable in C, we are given a chunk of memory that has:

- a *name*, in this case 'x'.
- a *value*, in this case, 7.
- a *location*, retrievable via '&x'
- a *type*, which tells us...
  - its *size* (you can get this in bytes using 'sizeof')
  - what we can do with it (e.g. z = x + y works with integers since c knows + is trying to add them)

Though not all the time will a chunk of memory have all these fields. When might this occur?

Revision (13) Pointers & Malloc cont.

What happens when we declare variables in C?

int x = 7;

So when we declare a variable in C, we are given a chunk of memory that has:

- a *name*, in this case 'x'.
- a *value*, in this case, 7.
- a *location*, retrievable via '&x'
- a *type*, which tells us...
  - its *size* (you can get this in bytes using 'sizeof')
  - what we can do with it (we will take about this now)

Though not all the time will a chunk of memory have all these fields. When might this occur?

Correct! When you do a 'malloc'

Revision (14) Pointers & Malloc cont.

What do malloc's do?

int *p;

p = malloc(sizeof(int));

What does the code above do?

Revision (15) Pointers & Malloc cont.

What do malloc's do?

int *p;

p = malloc(sizeof(int));

What does the code above do?

A malloc function takes in a size (in bytes) and then returns you the address or *location* of that memory chunk. So here, p stores an address to the memory chunk we have just asked malloc to give us.

Revision (16) Pointers & Malloc cont.

What do malloc's do?

int *p;

p = malloc(sizeof(int));

What does the code above do?

A malloc function takes in a size (in bytes) and then returns you the address or *location* of that memory chunk. So here, p stores an address to the memory chunk we have just asked malloc to give us.

Revision (17) Pointers & Malloc cont.

What do malloc's do?

int *p;

p = malloc(sizeof(int));

What does the code above do?

A malloc function takes in a size (in bytes) and then returns you the address or *location* of that memory chunk. So here, p stores an address to the memory chunk we have just asked malloc to give us.

What is this declartion int *p? Whenever we see a declaration with a star, like int* x or char *y what does it mean?

Revision (18) Pointers & Malloc cont.

What do malloc's do?

int *p;

p = malloc(sizeof(int));

What does the code above do?

A malloc function takes in a size (in bytes) and then returns you the address or *location* of that memory chunk. So here, p stores an address to the memory chunk we have just asked malloc to give us.

What is this declartion int *p? Whenever we see a declaration with a star, like int* x or char *y what does it mean?

It means that when I do 'type* var', 'var' is an address or *location* of a memory chunk holding that 'type'.

Revision (19) Pointers & Malloc cont.

What do malloc's do?

int *p;

p = malloc(sizeof(int));

How do I actually use pointers? How do I store a number into the memory p points to?

Revision (20) Pointers & Malloc cont.

What do malloc's do?

int *p;

p = malloc(sizeof(int));

How do I actually use pointers? How do I store a number into the memory p points to?

*p = 10

Nice! What does this do? We can think of * as the opposite of &.

&x: Converts variable *name* to *location*

p: Converts variable location* to *name*

So we can assign to the variable in that location, the value 10.

Revision (21) Pointers & Malloc cont.

What do malloc's do?

int *p;

p = malloc(sizeof(int));

How do I actually use pointers? How do I store a number into the memory p points to?

*p = 10

Now what is the correct print statement to print 10?

a: printf("%d", p);

b: printf("%d", &p);

Revision (22) Pointers & Malloc cont.

What do malloc's do?

int *p;

p = malloc(sizeof(int));

How do I actually use pointers? How do I store a number into the memory p points to?

*p = 10

Now what is the correct print statement to print 10?

a: printf("%d", p); (Correct)

b: printf("%d", &p); (Wrong)

Remember that printf requires an address. 'p' is already an address to the variable that is 10 and therefore, we only need to give it p.

Revision (22) Types 2.

So when we declare a variable in C, we are given a chunk of memory that has:

- a *name*, in this case 'x'.
- a *value*, in this case, 7.
- a *location*, retrievable via '&x'
- a *type*, which tells us...
  - its *size* (you can get this in bytes using 'sizeof')
  - what we can do with it (What about this)

Revision (23) Types 2 cont.

Suppose I have some code:

'int *x = malloc(sizeof(int));

int *y = malloc(sizeof(int));

*x = 5

*y = 10

z = x + y'

What is wrong with this code?

Revision (24) Types 2 cont.

Suppose I have some code:

'int *x = malloc(sizeof(int));

int *y = malloc(sizeof(int));

*x = 5

*y = 10

z = x + y'

What is wrong with this code?

Correct, you can't add pointers together! Your compiler will warn you because '+' is not defined on pointers. Therefore, the type of a variable will also inform you of the operations you can do on it.

Revision (25) Interlude

Let's have a quick break to allow you to absorb that.

Come up and ask any questions you have.

Revision (26) Exercises

Suppose I have the following code:

int x = 7;

int *p = malloc(sizeof(int));

*p = 15

#printf("%d", &x); #What happens?

#printf("%d", p); #What happens?

#how do I set x using the value p points to.

#how do I set the variable p points to with x.

Revision (27) Exercises

Suppose I have the following code:

int x = 7;

int *p = malloc(sizeof(int));

*p = 15

#printf("%d", &x); #What happens? (7)

#printf("%d", p); #What happens? (15)

#how do I set x using the value p points to. (x = *p)

#how do I set the variable p points to with x. (*p = x)

Revision (28) Exercises

Suppose I have the following code:

int x = 7;

int *p = malloc(sizeof(int));

*p = 15

p = &x

*p = 15

#printf("%d", &x); #What happens?

Revision (29) Exercises

Suppose I have the following code:

int x = 7;

int *p = malloc(sizeof(int));

*p = 15

p = &x

*p = 15

#printf("%d", &x); #What happens? (15)

Revision (30) Exercises

Suppose I have the following code:

int x = 7;

int *p = malloc(sizeof(int));

*p = 15

p = &x

p = 15

#printf("%d", &x); #What happens?

#printf("%d", *p); #What happens?

Revision (31) Exercises

Suppose I have the following code:

int x = 7;

int *p = malloc(sizeof(int));

*p = 15

p = &x

p = 15

#printf("%d", &x); #What happens? (7)

#printf("%d", *p); #What happens? (Whatever value is at address 15, some random trash left there)

Revision (32) Scope

A C program sees data as a collection of *variables*

Variables are examples of *computational objects*

Each computational object has

- a *location* in memory (obtainable via &)
- a *value* (ultimately just a bit-string)
- a *name* (unless created by `malloc()`)
- a *type*, which determines ...
  - its *size* (in units of whole bytes, `sizeof`)
  - how to *interpret* its value; what *operations* apply
- a *scope* (where it's visible within the program)
- a *lifetime* (during which part of program execution it exists)

Revision (32) Scope

Suppose we have the code if (*some condition*) { int x = 10; } printf("%d", &x);

Is this allowed?

Revision (33) Scope

Suppose we have the code if (*some condition*) { int x = 10; } printf("%d", &x);

Is this allowed? No! C has concepts of scopes. If x is declared in the *if* statement, then we cannot access it outside. Scopes are complex, but hopefully the compiler will tell you when you are breaking rules and you'll get a feel for it with time.

Memory: The C View of Data

A C program sees data as a collection of *variables*

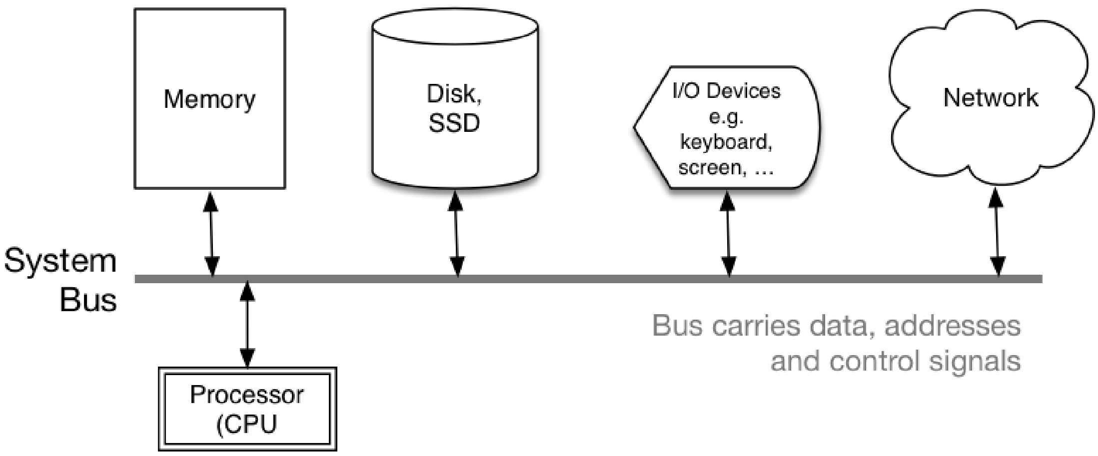
Variables are examples of *computational objects*

Each computational object has

- a *location* in memory (obtainable via &)
- a *value* (ultimately just a bit-string)
- a *name* (unless created by `malloc()`)
- a *type*, which determines ...
  - its *size* (in units of whole bytes, `sizeof`)
  - how to *interpret* its value; what *operations* apply
- a *scope* (where it's visible within the program)
- a *lifetime* (during which part of program execution it exists)

Computer Systems

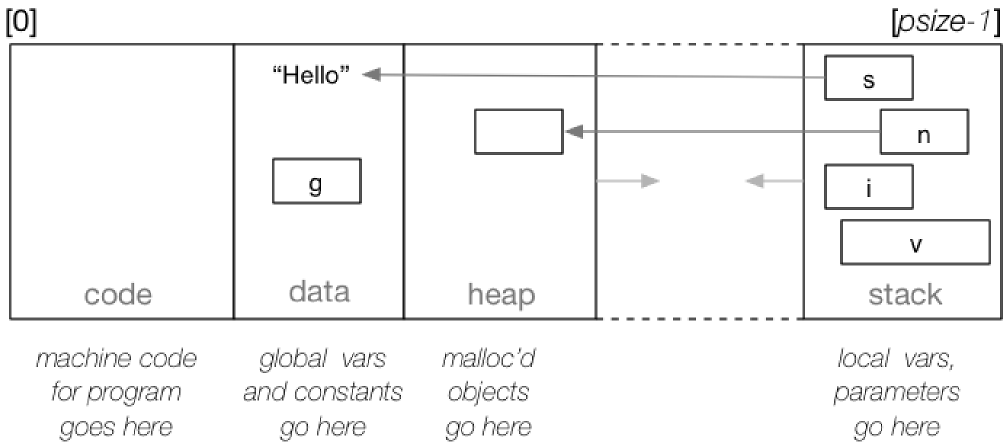
Component view of typical modern computer system



C: Runtime memory Usage

Run-time memory usage depends on language processor.

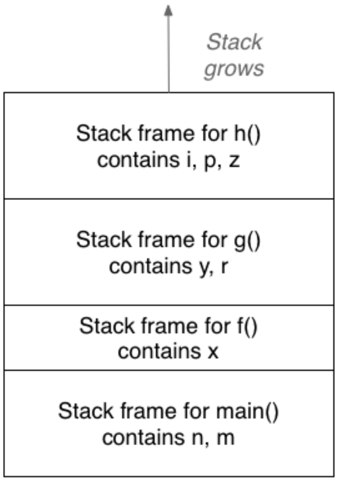
Memory regions during C program execution ...



C: Runtime Stack Usage

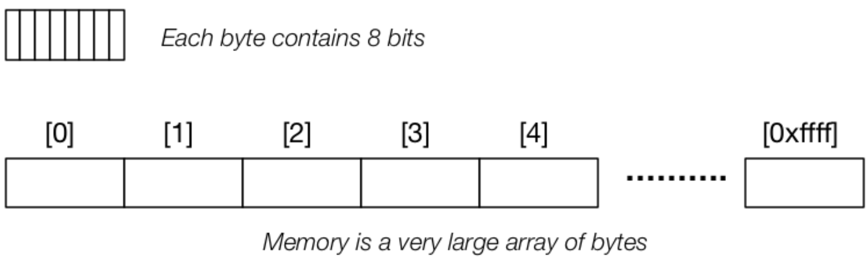
Example of runtime stack during call to h()

```
int main() {
   int n, m;
   n = 5;  m = f(n);
}
int f(int x) {
   return g(x);
}
int g(int y) {
   int r = 4 * h(y);
   return r;
}
int h(int z) {
   int i, p = 1;
   for (i=1; i<=z; i++)
      p = p * i;
   return p
}
```



The Physical View of Data

Memory = indexed array of bytes



Indexes are “memory addresses” (a.k.a. pointers)

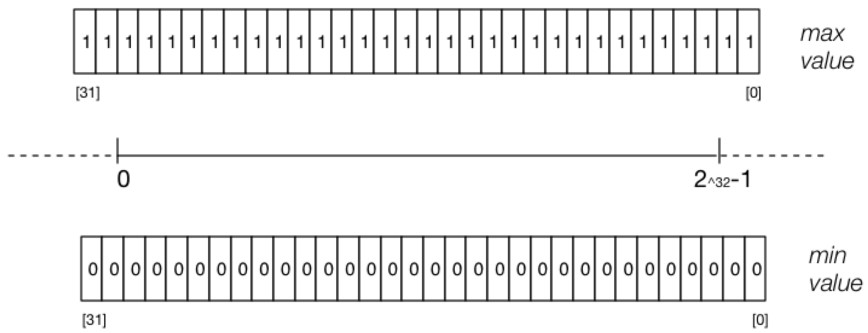
Properties of physical memory

- called main memory (or RAM, or primary storage, ...)
- indexes are “memory addresses” (a.k.a. pointers)
- data can be fetched in chunks of 1,2,4,8 bytes
- cost of fetching any byte is same (ns)
- usually volatile
- when addressing objects in memory ...
  - any byte address can be used to fetch 1-byte object
  - byte address for N-byte object must be divisible by N

Unsigned integers

The `unsigned int` data type

- commonly 32 bits, storing values in the range 0 .. 2^{32}-1



Decimal Representation

- Can interpret decimal number `4705` as:
  $4 \times 10^3 + 7 \times 10^2 + 0 \times 10^1 + 5 \times 10^0$
- The *base* or *radix* is 10 ... digits `0` – `9`
- Place values:

...	1000	100	10	1
...	10^3	10^2	10^1	10^0

- Write number as 4705_{10}
  - Note use of subscript to denote base

Representation in Other Bases

- base 10 is an arbitrary choice
- can use any base
- e.g. could use base 7
- Place values:

...	343	49	7	1
...	7^3	7^2	7^1	7^0

- Write number as 1216_7 and interpret as:
  $1 \times 7^3 + 2 \times 7^2 + 1 \times 7^1 + 6 \times 7^0 == 454_{10}$

Binary Representation

- Modern computing uses binary numbers
  - because digital devices can easily produce high or low level voltages which can represent 1 or 0.
- The *base* or *radix* is 2
  Digits `0` and `1`
- Place values:

...	8	4	2	1
...	2^3	2^2	2^1	2^0

- Write number as 1011_2 and interpret as:
  $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 == 11_{10}$

Converting between Binary and Decimal

- Example: Convert 1101_2 to Decimal:
- Example: Convert 29 to Binary:

Hexadecimal Representation

- Binary numbers hard for humans to read — too many digits!
- Conversion to decimal awkward and hides bit values
- Solution: write numbers in hexadecimal!
- The *base* or *radix* is 16 ... digits `0`, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- Place values:

...	4096	256	16	1
...	16^3	16^2	16^1	16^0

- Write number as $3AF1_{16}$ and interpret as:
  $3 \times 16^3 + 10 \times 16^2 + 15 \times 16^1 + 1 \times 16^0 == 15089_{10}$
- in C, **`0x`** prefix denotes hexadecimal, e.g. **`0x3AF1`**

Octal & Binary C constants

- Octal (based 8) representation used to be popular for binary numbers
- Similar advantages to hexadecimal
- in C a leading **`0`** denotes octal, e.g. **`07563`**
- standard C doesn't have a way to write binary constants
- some C compilers let you write **`0b`**
  - OK to use **`0b`** in experimental code but don't use in important code

```
printf("%u", 0x2A);      // prints 42
printf("%u", 052);       // prints 42
printf("%u", 0b101010); // might compile and print 42
```


Binary Constants

In hexadecimal, each digit represents 4 bits

```
    0100 1000 1111 1010 1011 1100 1001 0111
0x     4    8    F    A    B    C    9    7
```

In octal, each digit represents 3 bits

```
    01 001 000 111 110 101 011 110 010 010 111
0    1   1   0   7   6   5   3   6   2   2   7
```

In binary, each digit represents 1 bit

```
    0b01001000111110101011110010010111
```


Converting between Binary and Hexadecimal

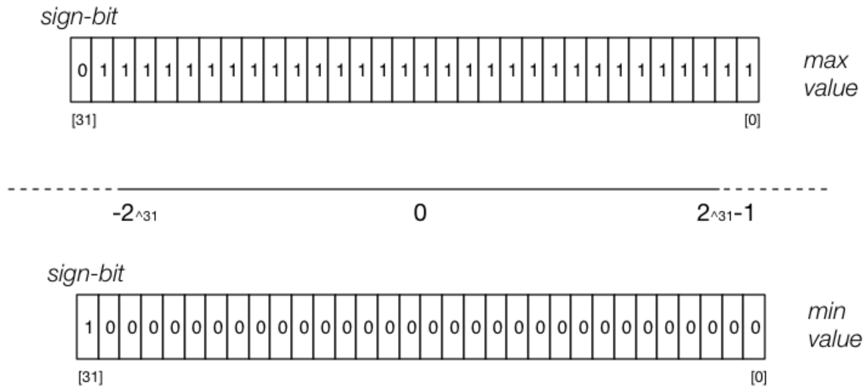
- Example: Convert 1011111000101001_2 to Hex:

- Example: Convert $1CED_{16}$ to Binary:

Signed integers

The `int` data type

commonly 32 bits, storing values in the range -2[31] .. 2[31]-1



Representing Negative Integers

- modern computers almost always use two's complement to represent integers
- positive integers and zero represented in obvious way
- negative integers represented in clever way to make arithmetic in silicon fast/simpler
- for an n-bit binary number the representation of $-b$ is $2^n - b$
- e.g. in 8-bit two's complement -5 is represented as $2^8 - 5$ == 11111011_2
- To form $-b$ from b you can also negate all then bits and then add 1
- e.g in 8-bit two's complement
  - 5 is represented as 00000101
  - If we negate all bits we get 11111010
  - If we then add 1 we get 11111011 which represents -5

Code example: printing all 8 bit twos complement bit patterns

- Some simple code to examine all 8 bit twos complement bit patterns.

```
for (int i = -128; i < 128; i++) {
    printf("%4d ", i);
    print_bits(i, 8);
    printf("\n");
}
```

source code for 8_bit_twos_complement.c

```
$ dcc 8_bit_twos_complement.c print_bits.c -o 8_bit_twos_complement
```

source code for print_bits.c source code for print_bits.h

Code example: printing all 8 bit twos complement bit patterns

```
$ ./8_bit_twos_complement
-128 10000000
-127 10000001
-126 10000010
...
  -3 11111101
  -2 11111110
  -1 11111111
   0 00000000
   1 00000001
   2 00000010
   3 00000011
...
 125 01111101
 126 01111110
 127 01111111
```


Code example: printing bits of int

```c
int a = 0;
printf("Enter an int: ");
scanf("%d", &a);
// sizeof returns number of bytes, a byte has 8 bits
int n_bits = 8 * sizeof a;
print_bits(a, n_bits);
printf("\n");
```

source code for print_bits_of_int.c

```
$ dcc print_bits_of_int.c print_bits.c -o print_bits_of_int
$ ./print_bits_of_int
Enter an int: 42
00000000000000000000000000101010
$ ./print_bits_of_int
Enter an int: -42
11111111111111111111111111010110
```


Code example: printing bits of int

```
$ ./print_bits_of_int
Enter an int: 0
00000000000000000000000000000000
$ ./print_bits_of_int
Enter an int: 1
00000000000000000000000000000001
$ ./print_bits_of_int
Enter an int: -1
11111111111111111111111111111111
$ ./print_bits_of_int
Enter an int: 2147483647
01111111111111111111111111111111
$ ./print_bits_of_int
Enter an int: -2147483648
10000000000000000000000000000000
$
```

Bits in Bytes in Words

- Many hardware operations works with bytes: 1 byte == 8 bits
- C's **`sizeof`** gives you number of bytes used for variable or type
- **`sizeof`** ***variable*** - returns number of bytes to store ***variable***
- **`sizeof (`*type*`)`** - returns number of bytes to store ***type***
- On CSE servers, C types have these sizes
  - `char` = 1 byte = 8 bits, 42 is 00101010
  - `short` = 2 bytes = 16 bits, 42 is 0000000000101010
  - `int` = 4 bytes = 32 bits, 42 is 00000000000000000000000000101010
  - `double` = 8 bytes = 64 bits, 42 = ?
- above are common sizes but not universal on a small embedded CPU
  `sizeof (int)` might be 2 (bytes)

Code example: `integer_types.c` - exploring integer types

We can use **`sizeof`** and **`limits.h`** to explore the range of values
which can be represented by standard C integer types **on our machine**...

```
$ dcc integer_types.c -o integer_types
$ ./integer_types
              Type Bytes Bits
              char     1    8
       signed char     1    8
     unsigned char     1    8
             short     2   16
    unsigned short     2   16
               int     4   32
      unsigned int     4   32
              long     8   64
     unsigned long     8   64
         long long     8   64
unsigned long long     8   64
```


Code example: `integer_types.c` - exploring integer types

```
              Type                  Min                  Max
              char                 -128                  127
       signed char                 -128                  127
     unsigned char                    0                  255
             short               -32768                32767
    unsigned short                    0                65535
               int          -2147483648           2147483647
      unsigned int                    0           4294967295
              long -9223372036854775808  9223372036854775807
     unsigned long                    0 18446744073709551615
         long long -9223372036854775808  9223372036854775807
unsigned long long                    0 18446744073709551615
```

source code for integer_types.c

stdint.h - integer types with guaranteed sizes

```
#include <stdint.h>
```

- to get below integer types (and more) with guaranteed sizes
- we will use these heavily in CP1521

```
              // range of values for type
              //             minimum               maximum
int8_t   i1; //                 -128                   127
uint8_t  i2; //                    0                   255
int16_t  i3; //               -32768                 32767
uint16_t i4; //                    0                 65535
int32_t  i5; //          -2147483648            2147483647
uint32_t i6; //                    0            4294967295
int64_t  i7; // -9223372036854775808   9223372036854775807
uint64_t i8; //                    0  18446744073709551615
```

source code for stdint.c

Code example: char_bug.c

Common C bug:

```
char c;  // c should be declared int   (int16_t would work, int is better)
while ((c = getchar()) != EOF) {
    putchar(c);
}
```

Typically stdio.h contains:

```
#define EOF -1
```

- most platforms: char is signed (-128..127)
  - loop will incorrectly exit for a byte containing 0xFF
- rare platforms: char is unsigned (0..255)
  - loop will never exit

source code for char_bug.c