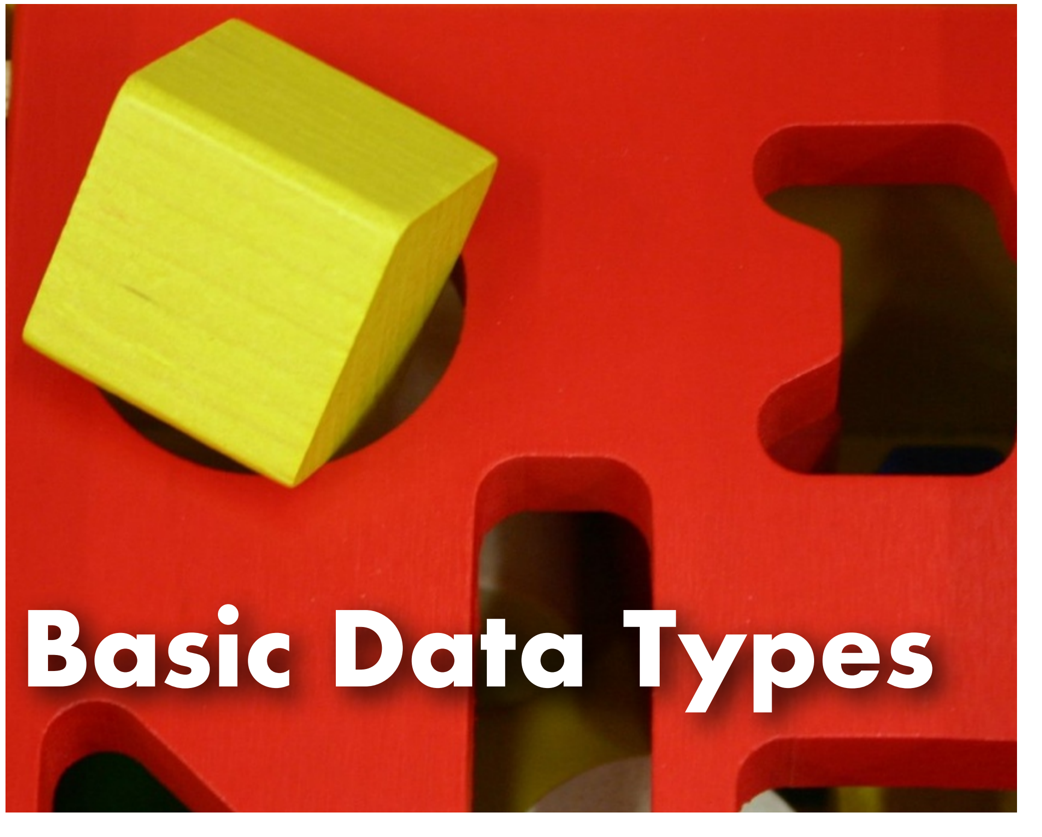
{C}

Lecture 3



Data Types

Data types are sets of values along with operations that manipulate them

For example, (signed) integers in C are made up of the set of values `…, -1, 0, 1, 2, …` along with operations such as addition, subtraction, multiplication, division…

Values must be mapped to data types provided by the hardware and operations compiled to sequences of hardware instructions

Types: Java v. C

In Java

- primitive types (int, float, char,...)
- object types (each object has a set of fields and methods)
- type conversion are checked at runtime to forbid nonsensical conversions, e.g., **`int`** to **`Object`**

In C, types have a less rigid definition

- are a convenient way of reasoning about memory layout
- all values (regardless of their type) have a common representation as a sequence of bytes in memory
- primitive type conversions are always legal

Data Types

Designing a computer language requires choosing which data types to build in, and which ones must be defined by users

The tradeoff is one of expressiveness vs. efficiency

Expressiveness refers to the ability to clearly express solutions to computational problem. *Abstraction*

Efficiency refers to the ability to map the data type's operation to machine instructions. *Performance*

The design of C typically favors efficiency

`pits` – semantics

Imagine a data type called **`pit`** (for **`p`**air **`o`**' **`bits`**) with values: `0, 1, 2, 3`

`pits` have one operation the postfix increment: `++`

The meaning of `++` is the function:

`0 ++ = 1,    1 ++ = 2,    2 ++ = 3,    3 ++ = 0`

Observations:

- The data type is finite
- The increment operation wraps around

pits – implementation

A variable of type `pit` is represented by two bits in memory

00 = 0

01 = 1

10 = 2

11 = 3

Pseudo code implementation of ++

```
pit pp(pit p) {
    int x = p;
    x += 1;
    if (x==4) x = 0;
    return (pit) x;
}
```

signed pits

To represent **signed pits** one bit is need for the sign:

- decrease the range of values
- increase the number of bits used to represent the data type

Assume the size of the data type must not change.

signed pits can take the following values: **-1, 0, 1**

The implementation is as follows

00 = 0   01 = 1   10 = 0   11 = -1

signed pits

00 = 0   01 = 1   10 = 0   11 = -1

00 ++ = 01, 01 ++ = 11, 10 ++ = 01, 11 ++ = 10

The increment function is somewhat more complex

```
signed pit function<++>(signed pit p) {
  int x = p;
  int s = (p >> 1);  /* get sign */
  if (x == 1) then return -1   /* max value */
    else {
      x++;
      if (s) then return x else x + 2
         /* incorporate sign bit */
    }
```

Byte

A byte = 8 bits

- Decimal 0 to 255
- Hexadecimal 00 to FF
- Binary 00000000 to 11111111

In C:

- Decimal constant: `12`
- Octal constant: `014`
- Hexadecimal constant: `0xC`

0_{hex}	=	0_{dec}	=	0_{oct}	0	0	0	0
1_{hex}	=	1_{dec}	=	1_{oct}	0	0	0	1
2_{hex}	=	2_{dec}	=	2_{oct}	0	0	1	0
3_{hex}	=	3_{dec}	=	3_{oct}	0	0	1	1
4_{hex}	=	4_{dec}	=	4_{oct}	0	1	0	0
5_{hex}	=	5_{dec}	=	5_{oct}	0	1	0	1
6_{hex}	=	6_{dec}	=	6_{oct}	0	1	1	0
7_{hex}	=	7_{dec}	=	7_{oct}	0	1	1	1
8_{hex}	=	8_{dec}	=	10_{oct}	1	0	0	0
9_{hex}	=	9_{dec}	=	11_{oct}	1	0	0	1
A_{hex}	=	10_{dec}	=	12_{oct}	1	0	1	0
B_{hex}	=	11_{dec}	=	13_{oct}	1	0	1	1
C_{hex}	=	12_{dec}	=	14_{oct}	1	1	0	0
D_{hex}	=	13_{dec}	=	15_{oct}	1	1	0	1
E_{hex}	=	14_{dec}	=	16_{oct}	1	1	1	0
F_{hex}	=	15_{dec}	=	17_{oct}	1	1	1	1

http://en.wikipedia.org/wiki/Hexadecimal

Words

Hardware has a `Word size` used to hold integers and addresses

The size of address words defines the maximum amount of memory that can be manipulated by a program

Two common options:

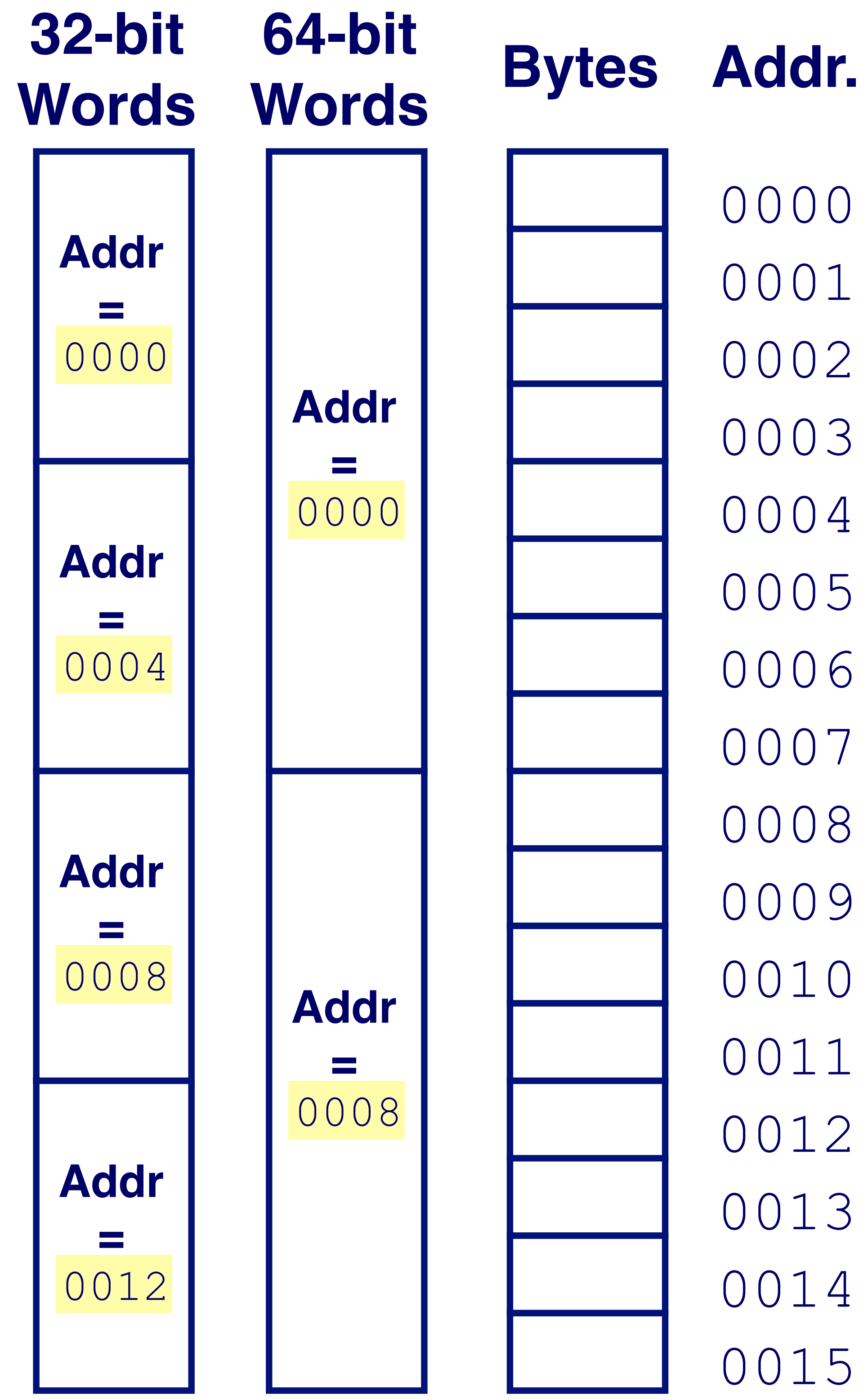
- 32-bit words => can address 4GB of data
- 64-bit words => could address up to 1.8×10^{19}

Different words sizes (integral number of bytes, multiples and fractions) are supported

Addresses

Addresses specify byte location in computer memory

- address of first byte in word
- address of following words differ by 4 (32-bit) and 8 (64-bit)



Data Types

The base data type in C

- **`int`** - used for integer numbers
- **`float`** - used for floating point numbers
- **`double`** - used for large floating point numbers
- **`char`** - used for characters
- **`void`** - used for functions without parameters or return value
- **`enum`** - used for enumerations

The composite types are

- pointers to other types
- functions with arguments types and a return type
- arrays of other types
- **`structs`** with fields of other types
- **`unions`** of several types

Qualifiers, Modifiers & Storage

Type specifiers

- `short` - decrease storage size
- `long` - increase storage size
- `signed` - request signed representation
- `unsigned` - request unsigned representation

Type qualifiers

- `volatile` - value may change without being written to by the program
- `const` - value not expected to change

Storage class

- `static` - variable that are global to the program
- `extern` - variables that are declared in another file

Sizes

Type	Range (32-bits)	Size in bytes
`signed char`	–128 to +127	1
`unsigned char`	0 to +255	1
`signed short int`	–32768 to +32767	2
`unsigned short int`	0 to +65535	2
`signed int`	–2147483648 to +2147483647	4
`unsigned int`	0 to +4294967295	4
`signed long int`	–2147483648 to +2147483647	4 or 8
`unsigned long int`	0 to +4294967295	4 or 8
`signed long long int`	–9223372036854775808 to +9223372036854775807	8
`unsigned long long int`	0 to +18446744073709551615	8
`float`	1×10^{-37} to 1×10^{37}	4
`double`	1×10^{-308} to 1×10^{308}	8
`long double`	1×10^{-308} to 1×10^{308}	8, 12, or 16

Character representation

ASCII code (American Standard Code for Information Interchange): defines 128 character codes (from 0 to 127),

In addition to the 128 standard ASCII codes there are other 128 that are known as extended ASCII, and that are platform-dependent.

Examples:

- The code for 'A' is 65
- The code for 'a' is 97
- The code for 'b' is 98
- The code for '0' is 48
- The code for '1' is 49

Understanding types matter...

Types define an abstraction or approximation of a computation....

More practically, there are implicit conversions that take place and they may result in truncation, and ...

Some data types are not interpreted the same on different platforms, they are machine-dependent

- **`sizeof( x )`** returns the size in bytes of the object **`x`** (either a variable or a type) on the current architecture

Declarations

The declaration of a variable allocates storage for that variable and can initialize it

```
int lower = 3, upper = 5;
char c = '\\', line[10], he[3] = "he";
float eps = 1.0e-5;
char arrdarr[10][10];
unsigned int x = 42;
char* ardar[10];
char* a;
void* v;
void foo(const char[]);
```

Without an explicit initializer local variables may contain random values (static & extern are zero initialized)

Conversions

What is the meaning of an operation with operands of different types?

```
char c; int i;   … i + c …
```

The compiler will attempt to convert data types without losing information; if not possible emit a warning and convert anyway

Conversions happen for operands, function arguments, return values and right-hand side of assignments.

Reading

K&R:

- Chapter 1, pp. 22 - 34
- Chapter 2, pp. 35 - 48