Programming in C

Data types: basic types, pointers, arrays, strings.

What are types?

- Data types are sets of values along with operations that manipulate them
- Values must be mapped to data types provided by the hardware and operations compiled to sequences of hardware instructions
- Example: integers in C are made up of the set of values **..., -1, 0, 1, 2, ...** along with operations such as addition, subtraction, multiplication, division...

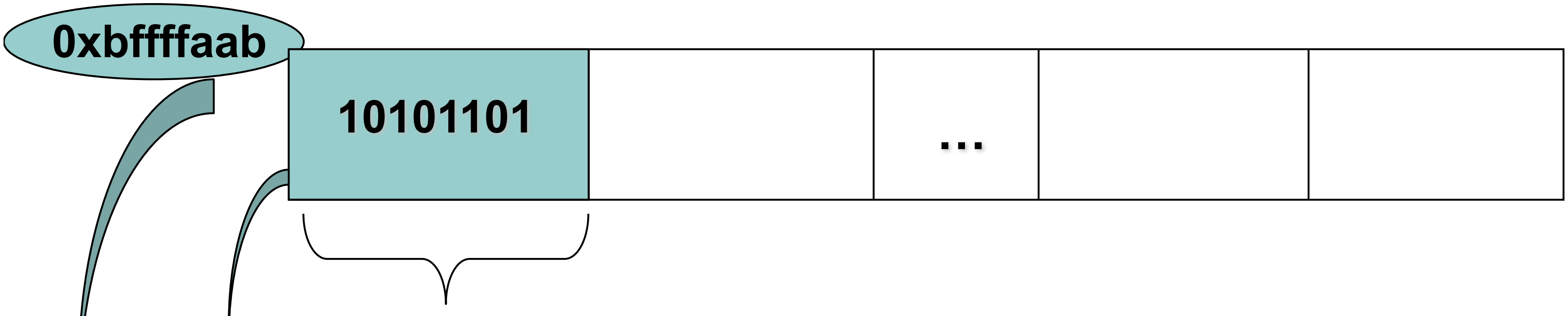
Types in C

- 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

Words

- Hardware has a `**<u>Word size</u>**` used to hold integers and addresses
  - Different words sizes (integral number of bytes) are supported
  - Modern general purpose computers usually use 32 or 64 bits
- The size of address words defines the maximum amount of memory that can be manipulated by a program
  - 32-bit words => can address 4GB of data
  - 64-bit words => could address up to 1.8 x 10^{19}

Addresses



8 bits = byte (2^8 = 256 different values)

- Addresses specify byte location in computer memory, i.e. address of first byte in a word
- Address of following words differ by 4 (32-bit) and 8 (64-bit)

Types representation

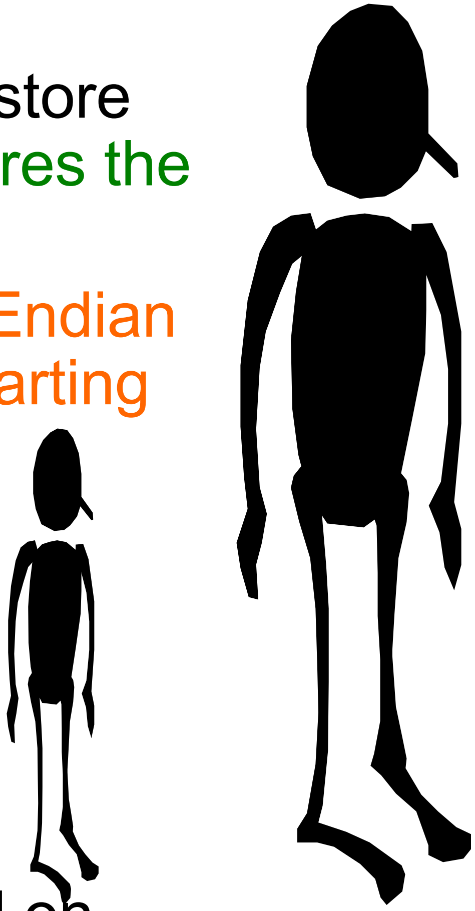
- **Basic types**
  - int - used for integer numbers
  - float - used for floating point numbers
  - double - used for large floating point numbers
  - char - used for characters
  - void - used for functions without parameters or return value
  - enum - used for enumerations
- **Composite types**
  - 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 qualifiers
  - short - decrease storage size
  - long - increase storage size
  - signed - request signed representation
  - unsigned - request unsigned representation
- Type modifiers
  - 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

Byte order

- Different systems store multibyte values (for example int) in different ways.
  - HP, Motorola 68000, and SUN systems store multibyte values in Big Endian order: stores the high-order byte at the starting address
  - Intel 80x86 systems store them in Little Endian order: stores the low-order byte at the starting address.
- Data is interpreted differently on different hosts.
- Where it shows up:
  - Network protocols
  - Binary file created on a computer is read on another computer with different endianness.



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

sizeof(x) returns the size in bytes.

Characters 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 ‘0’ is 48

Understanding types in C matters …

- Incorrect use may result in bugs
  - There are implicit conversions that take place and they may result in truncation
  - Some data types are not interpreted the same way 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
- Ineffective use may result in higher cost
  - Storage, performance

What will this program output?

```
#include <stdio.h>
int main() {
   char c = -5;
   unsigned char uc = -5;

   printf(”%d  %d  \n", c, uc);

   return 0;
}
```

Printf format

c	Character
d or i	Signed decimal integer
f	Decimal floating point
s	String of characters
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
p	Pointer address

NOTE: read printf man pages for additional formats

What will this program output?

```
#include <stdio.h>
int main() {
   char c = ‘a’;

   printf(“%c  %d  %x \n", c, c, c);

   return 0;
}
```

```
#include <stdio.h>

int main() {
  char           c;
  short int      s_i;
  long int       l_i;
  int            i;
  float          f;
  double         d;
  long double    l_d;

  printf(" Size of char:        %d (bytes)\n", sizeof(c));
  printf(" Size of short:       %d (bytes)\n", sizeof(s_i));
  printf(" Size of long:        %d (bytes)\n", sizeof(l_i));
  printf(" Size of int:         %d (bytes)\n", sizeof(i));
  printf(" Size of float:       %d (bytes)\n", sizeof(f));
  printf(" Size of double:      %d (bytes)\n", sizeof(d));
  printf(" Size of long double: %d (bytes)\n", sizeof(l_d));

  return 0;
}
```

Implicit conversions: What can go wrong?

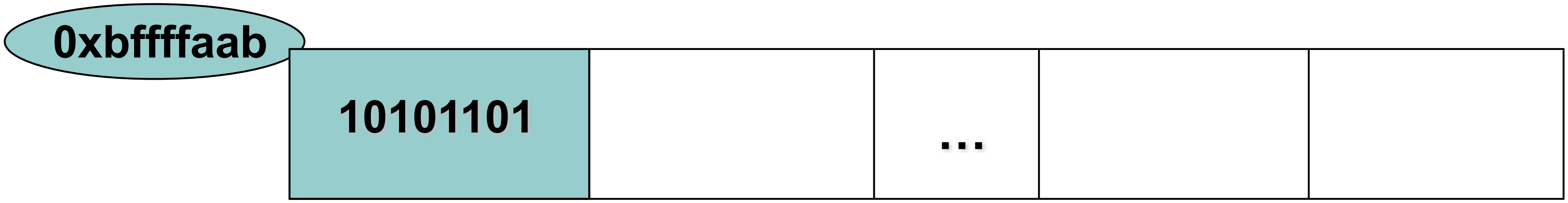
```
#include <stdio.h>
int main () {
   short s = 9;
   long  l = 32770;
   printf("%d\n", s);
    s = l;
    printf("%d\n", s);

    return 0;
}
```

short can store -32768 to 32767

Pointers

- The address of a location in memory is also a type based on what is stored at that memory location
  - `char *` is “a pointer to char” or the address of memory where a char is stored
  - `int *` points to a location in memory where a int is stored
  - `float *` points to a location in memory where a float is stored
- We can do operations with this addresses
- The size of an address is platform dependent.



& and *

- Given a variable v

  `&v` means the address of v

- Given a pointer ptr

  `*ptr` means the value stored at the address specified by ptr

```
#include <stdio.h>

int main() {
  char c;
  char *c_ptr = &c;

  printf(“Size of char *:    %d (bytes)\n", sizeof(c_ptr));
  printf(“Address of c is:   %p  \n", &c);
  printf(“Value of c_ptr is: %p  \n", c_ptr);

   printf(“Value of c is:     %c  \n", c);
   printf(“Value of *c_ptr is:%c  \n", *c_ptr);

  return 0;
}
```

Arrays of characters

```
char c[10];

for (i=0; i< 10; i++) {
   printf("%c\n", c);
}
```

`&c[0] or c` (the name of the array) represents the start memory address where the array is stored in the memory

char *p = &c[0];

First element of the array starts at index 0, in this case c[0]

Arrays of characters

```
char c[10];
char *p = &c[0];

for (i=0; i < 10; i++) {
  c[i] = ‘a’;
}
c[5] = ‘b’;
```

What’s the address of c[5]? It is p+5

Pointer vs. what’s stored at the address indicated by a pointer

```
#include <stdio.h>

int main() {
  char    c;
  char *  c_ptr = &c;
  char    array[5];

  array[2] = 'b';
  c_ptr = array;

  printf("Address where array starts:              %p\n", array);
  printf("Value of variable c_ptr:                 %p\n", c_ptr);
  printf("Value stored at the address c_ptr+2:     %c\n", *(c_ptr+2));

  return 0;
}
```

Strings

- In C a string is stored as an array of characters, terminated with null, 0, hex 00 or ‘\0’
- The array has to have space for null
- Function strlen returns the length of the string excluding the string terminator

ALWAYS MAKE SURE YOU DON’ T GO BEYOND THE SIZE OF THE ARRAY – 1; the last item in the array should be the null string terminator

Symbolic constants: #define

- Followed by the name of the macro and the token sequence it abbreviates
- By convention, macro names are written in uppercase.
- There is no restriction on what can go in a macro body provided it decomposes into valid preprocessing tokens.
- If the expansion of a macro contains its name, it is not expanded again
- **`#define NO100`**

#define vs const modifier

- Declaring some variable with const means that its value can not be modified
- `const int no = 100;`
- Alternative is to use #define
- `#define NO 100`

- Is there any difference?

```
#include<stdio.h>

const int MAX=10;

int main() {
      char s[MAX];
       int i;

       s[MAX-1] = 0;

        for(i=0; i<MAX-1; i++) {
              s[i] = ‘a’;
        }

   s[0] = ‘b’;
    printf("%s\n", s);

    return 0;
}
```

What’s wrong with this code?

Consider that we have the following declaration

```
const int MAX=10;
int main() {
char s[MAX];
```

….

What’s wrong in each of the following:

```
(1)      s[MAX] = 0;

(2)
        for(i=1; i<=MAX; i++) {
              s[i] = ‘a’;
        }
         printf("%s\n", s);

(3) MAX = 12;
```

Strlen vs sizeof

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

const int MAX = 10;
int main() {
  char s[MAX];
  int len, size, i;

  s[0] = 'a';
  s[1] = '\0';

  len = strlen(s);
  size = sizeof(s);

  printf("len: %d characters, size: %d bytes\n", len, size);
  printf("The content of array s is: ");
  for(i=0; i< MAX; i++) {
    printf("%X  ", s[i]);
  }

  printf("\n");

  return 0;
}
```

Operations with strings

- **strlen**
- **strncpy vs strcpy**
- **strncmp vs strcmp**
- **/usr/include/string.h**

```
int strlen(char s[]) {
  int i = 0;
  while(s[i] != ‘\0’)
      ++i;
   return i;
}
```

Good coding habits

- **Use const and or define for SIZES and avoid using numbers in the code**
- **Always check your arrays, that they start at 0 and end at SIZE-1**
- **Allow space for null in strings**

Boolean

- Std 89 the first C standard does not define boolean
- It I supported in standard std 99.
- It is not really a needed type and that’s why was not included in the original design
- #include <stdbool.h> type is _Bool

Readings for this lecture

K&R Chapter 1 and 2

READ man for printf

http://en.wikipedia.org/wiki/Word_(computer_architecture)

READ string related functions