CS240: Programming in C

Lecture 3: More on Types

Type representation and enforcement

```
#include <stdio.h>
int main () {
   short s = 9;
   long  l = 32768;

   printf("%d\n", s);
    s = l;
    printf("%d\n", s);

    return 0;
}
```

short can store -32768 to 32767

Type representation and enforcement

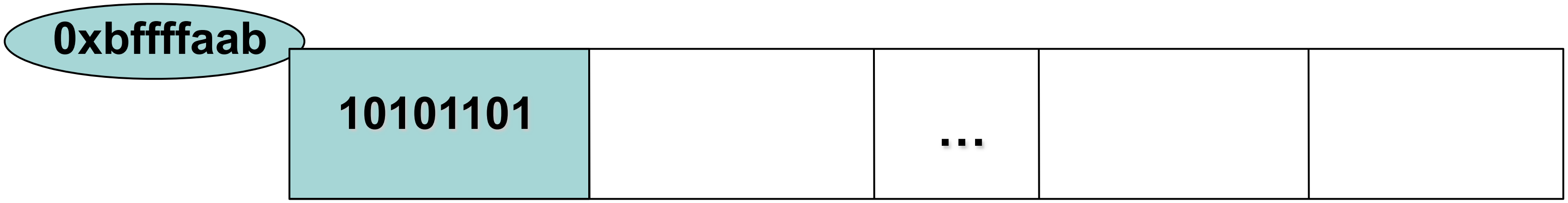
```
#include <stdio.h>
int main () {
   short s = 9;
   long  l = 32768;

   printf("%d\n", s);
    s = l;
    printf("%d\n", s);

    return 0;
}
```

short can store -32768 to 32767

-32768

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 operation with this addresses
- The size of an address is platform dependent, on many is 32 bits.



0xbffffaab

10101101

...

& 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

All variables are associated with an address (a memory location). Both the contents of the location as well as the location itself are manipulable.

Example

```
#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);


  return 0;
}
```

Example

```
#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);

  return 0;
}
```

```
Size of char *:    8 (bytes)
Address of c is:   0x7fff5fbff62f
Value of c_ptr is: 0x7fff5fbff62f
```

Arrays of characters

```
char c[10];

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

`&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];
char t;

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

What’s the address of `c[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 start:               %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;
}
```

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

```
Address where array start:              0x7fff5fbff620
Value of variable c_ptr:                0x7fff5fbff620
Value stored at the address c_ptr+2:    b
```

```
#include <stdio.h>

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

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

  printf("Address where array start:              %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;
}
```

Constant variables

- 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?
  - #define is a macro -- evaluated at compile-time
  - can you declare a pointer to a constant? to a #define?
  - typechecking?
- **`enum boolean{ NO, YES};`**

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

Example

```
#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;
}
```

Example

```
#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;
}
```

```
baaaaaaaa
```

What’s wrong with this code?

Consider 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;
```

```
baaaaaaaaaÀ_ÿ
```

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;
}
```

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;
}
```

```
len: 1 characters, size: 10 bytes
The content of array s is: 61  0  FFFFFFBF  5F  FFFFFFFF  7F  0  0  60  7
```

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;
}
```

Why can’t we simply copy strings using assignment?

s = t

“Incompatible types in assignment”

Readings for this lecture

K&R Chapter 1 and 2

READ string related functions