System SW

Lectures 3-4 – Basics of C programming language – Part 3

Jarno Tuominen / Reviewed by Sanna Määttä

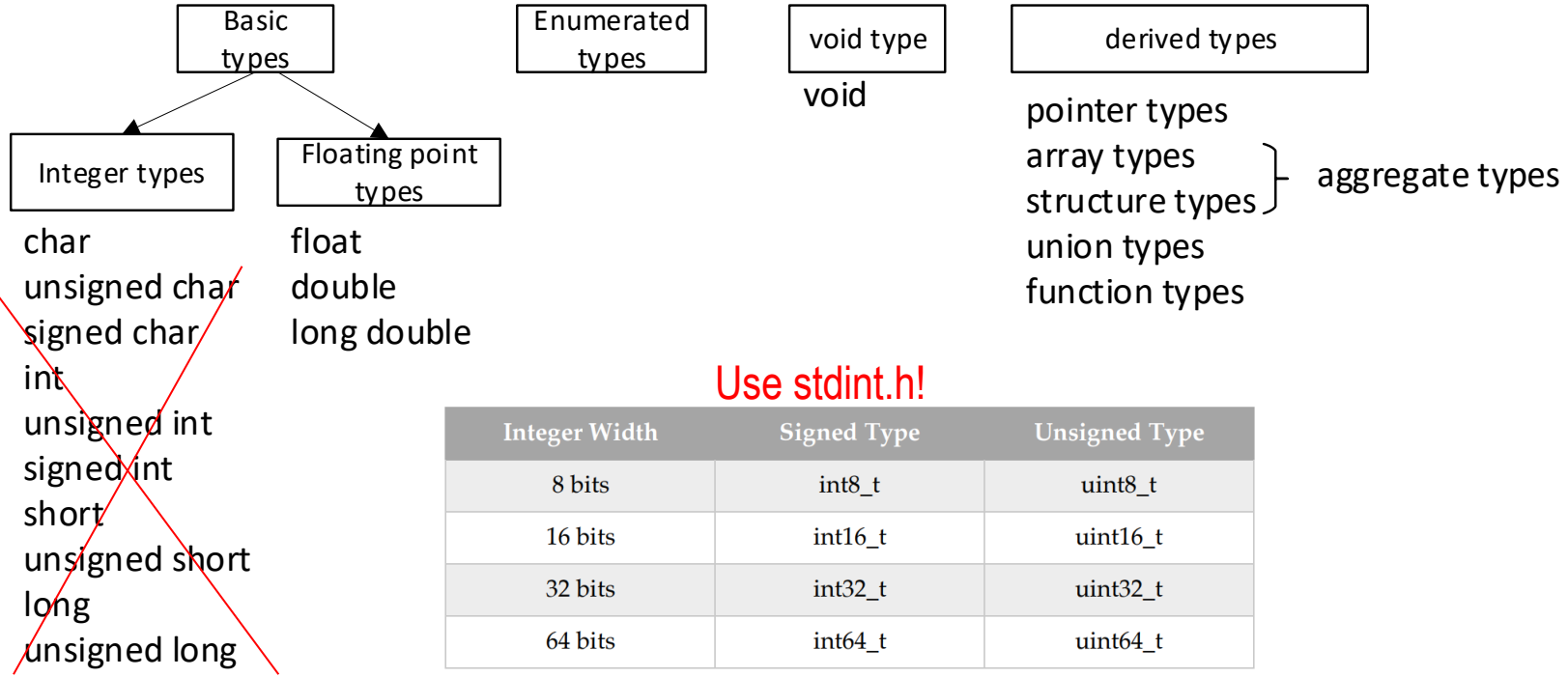
Lectures 3-4 – Basics of C programming language – Part 3

- Review
- Data types (cont)
- Type qualifiers
- Storage classes
- Type casting
- Pointers

Review of last lecture

- Basic data types
- Derived data types

Review: The data types in C



Use stdint.h!

Integer Width	Signed Type	Unsigned Type
8 bits	int8_t	uint8_t
16 bits	int16_t	uint16_t
32 bits	int32_t	uint32_t
64 bits	int64_t	uint64_t

Review: Enumeration

```
typedef enum {
        STATE_INIT = 0,
        STATE_SLEEP, //1
        STATE_IDLE, //2
        STATE_SETUP, //3
        STATE_SESSION_ACTIVE, //4
        STATE_PREVIEW, //5
        STATE_LOGGING = 10 , //10
        STATE_STOP_LOGGING, //11
        STATE_ERROR, //12
        STATE_POWER_OFF //13
} state_t;
```

Integer value of the first member (optional as 0 is the default)

The rest get values automatically

A value can be explicitely set, after which auto-increment continues

```
state_t state_next = STATE_INIT;

switch (state_next) {
        case STATE_ERROR:
        ...
```

- Enumerated types are **arithmetic** types
- Used to define variables that can only have **certain discrete integer values** throughout the program
- Makes your code more human-readable

Review: Structures

- Structures aggregate the storage of **multiple** data items, of potentially **differing** data types, into one memory block referenced by a single variable
- To define a structure, use struct statement
- Element selection with **member access operator** ”.”

Structure tag (optional)

members

```
struct Book_t {
   char title[50];
   int book_id;
};
```

Declare variable ”book” of type Book_t

```
int main() {
   struct Book_t book;
   strcpy( book.title, "C Programming");
   book.book_id = 6495407;

   printf( "Book title : %s\n", book.title);
   printf( "Book book_id : %d\n",
book.book_id);
   return 0;
}
```

Review: Union data types

- Like structures, union is a user defined data type
- In union, all members share the **same** memory location (in struct, each member has its own location)
- The members can be of different data type
- **Only one member can contain a value at any given time**
- The size of the union is defined by the largest member in union
  - Don't need to worry about packing
- Definition exactly like with structures, but use keyword `union` instead.

```
union data_t {
    int i;
    float f;
    char str[20];
} data;
```

```
typedef union {
  int i;
  float f;
  char str[20];
} data_t;
data_t data;
```

Review: Structure packing

- Storage for the basic C datatypes on an x86 or ARM processor doesn't normally start at arbitrary byte addresses in memory
  - Basic C types (incl pointers) on x86 and ARM are *self-aligned*
- A struct instance will have the alignment of its **widest** scalar member
- The "holes" are filled with **padding**, effectively increasing the storage size of the struct -> waste of memory
- By **repacking** the structure in a smart way, considerable memory savings possible!

Original

```
struct bad_one {
   char c; /* 1 byte */
   int y *p; /* 8 bytes */
   short x; /* 2 bytes */
};                                   11B
Becomes...
struct bad_one {
   char c;          /* 1 byte */
   char pad1[7];    /* 7 bytes */
   int y *p;        /* 8 bytes */
   short x;         /* 2 bytes */
   char pad2[6];    /* 6 bytes */
};                                   24B
```

Repacked

```
struct good_one {
   int y *p;
   short x;
   char c;                           11B
};
Becomes...
struct good_one {
   int y *p;        /* 8 bytes */
   short x;         /* 2 bytes */
   char c;          /* 1 byte */
   char pad[5];     /* 5 bytes */
};                                   16B
```

http://www.catb.org/esr/structure-packing/

Review: Bit fields

C99

```
#include <stdbool.h>
typedef struct {
  bool a;
  bool b;
  bool c;
} flags_t;

typedef struct {
  bool a: 1;
  bool b: 1;
  bool c: 1;
} flags_bits_t;
```

Up to 8 boolean flags can be packed to a single byte

- Using bit fields lead to huge memory savings!

```
Size of flags_t is 3 bytes
Size of flags_bits_t is 1 bytes
```

```
void main() {
  printf ("Size of flags_t is %d bytes\n", sizeof (flags_t));
  printf ("Size of flags_bits_t is %d bytes\n", sizeof (flags_bits_t));
}
```

Lecture 6 – Basics of C programming language – Part 3

- Review
- Data types (cont)
- Type qualifiers
- Storage classes
- Type casting
- Pointers

Arrays

- An array is a container for data which is of all the **same** type
- The data type can be a basic type (integer, char etc) or a derived type (struct, pointer, another array etc)
- Defined by square brackets **[ ]**
- Static arrays: The size of an array is known at the compile time
- Dynamic arrays involve the use of pointers and malloc() to allocate memory from the heap memory area (will be covered later on)

```
//Declaration
type name[number of elements];
int numbers[6];

//Declaration + initialization
type name[number of elements]=
{comma-separated values};

int point[6]={0,0,1,0,0,0};

//When initializing at declaration,
//size can be omitted
int point[]={0,0,1,0,0,0};

//Multi-dimensional arrays
int two_d[2][3];

int two_d[][] = {
          { 5, 2, 1 },
          { 6, 7, 8 }
};
```

Strings in C

- There is no string type in C – instead, strings are stored in **character arrays**
- A string is **terminated** by a null- character '\0', which is automatically added to the end by compiler in the first example
- In the latter example null-character needs to be inserted manually
- All the string library functions expect strings to be terminated by a null-character
- String indexing starts from zero
- String handling routines are in string library <string.h> (along with very important memory handling routines)

```
char string[5] = "Blaa";

char string[5] = {'B','l','a','a','\0'};
```

0	1	2	3	4
B	l	a	a	\0

```
#include <stdio.h>
int main()
{
    char string1[5] = "Blaa";
    char string2[5] = {'B','l','a','a','\0'};
    char string3[4] = "Blaa"; //no!!!
    printf("%s\n",string1);
    printf("%s\n",string2);
    printf("%s\n",string3);
    return 0;
}
```

```
Blaa
Blaa
Blaa
```

← Oops!

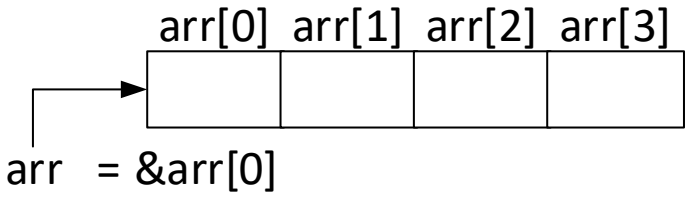
https://www.tutorialspoint.com/c_standard_library/string_h.htm

Relation between arrays and pointers

- C does not have array variables (or string variables, which are char arrays)
- In fact, the array notation **[]** is actually an alternate way of using pointers
- Consider an array: int arr[4]
- Name of the array always points to the first element [0] of array
- This means that arr is equivalent to &arr[0]
- Futhermore, statement

  arr[2] =5; is identical to

  *(arr+2) = 5;
- So, arr is a pointer!



We will revisit this topic soon...

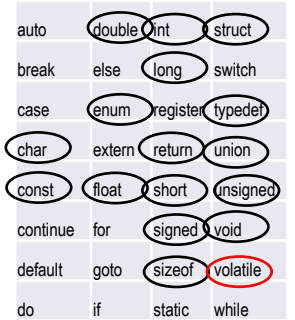
Lecture 6 – Basics of C programming language – Part 3

- Review
- Data types (cont)
- Type qualifiers
- Storage classes
- Type casting
- Pointers

Type qualifiers

- All variables are considered ”unqualified” by default
- Type qualifiers modify the properties of variables in certain ways
- C has 4 type qualifiers:
  - const (C89, already covered)
  - volatile (C89)
    - Tells the compiler that the value of the variable can be changed anytime – outside from the current context. Typically used with variables which are used in interrupt handlers. In practice, this is a directive to compiler NOT to optimize it.
    - For example two consecutive reads of a same variable is typically optimized away – but what if there is an interrupt between the reads? The value would be different. Thus, NO optimization as it would break your program.
  - restrict (C99, related to pointers & optimization, skipped)
  - _Atomic (C11, skipped)

https://www.bezem.de/pdf/tqinc.pdf



Lecture 6 – Basics of C programming language – Part 3

- Review
- Data types (cont)
- Type qualifiers
- Storage classes
- Type casting
- Pointers

Storage classes of variables and functions

- Storage Classes are used to describe about the features of a variable/function. These features include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.

- Every variable and function in C has one of the following storage classes:

  static
  functions can be called only from the same .c-file
  variables preserve their value even after they are out of scope

  auto - local variables have local lifetime. (default, so you don't have to use this ever)

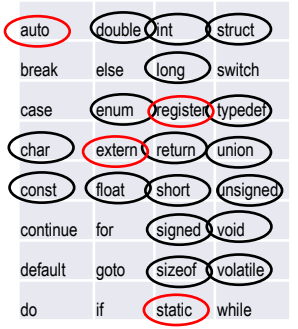
  extern - tells the compiler that the variable is defined elsewhere

  register - a hint to compiler to cache the variable in a processors register.
  Avoid, if you are not smarter than compiler

- If the declaration does not specify the storage class, a context-dependent default is used:

  extern for all top-level declarations in a source file (i.e. globals)
  auto for variables declared in function bodies (i.e. locals)

  RULE: don't use keyword **auto** – it's completely useless
  RULE: use keyword **static** when needed, but avoid overuse
  (Barr: C coding standard 2018)



Static variables

- static keyword has two meanings, depending on **where** the static variable is declared
- **Outside a function**, static variables/functions are only visible within that .c-file where it is declared (file scope), not globally. Static variable cannot be extern'ed. This could be thought as some level of "safety".
- **Inside a function**, static variables:
  - are still **local** to that function
  - are **initialized only once** during program initialization (do not get reinitialized with each function call)

```c
#include <stdio.h>

int count_up(void) {
    static int counter = 0; //initialized only once
    //int counter = 0; //non-static would not work!
    counter++;
    if ( (counter%10) == 0) {
        printf("got 10 hits!\n");
    }
    return counter; //so that we can read the counter
}

int main() {
    while(1) {
        if (count_up() > 100) {
            printf("100 is enough...\n");
            break;
        }
    }
    return 0;
}
```

External variables

- By default, all the global variables (defined outside functions) are external
- extern keyword tells the compiler that the variable is defined elsewhere
- The main purpose of using extern is to make the variable visible across multiple files/modules
- Assume we have a buffer (fifo), to which sensor data is collected and once it is almost full, it's contents is flushed to a memory card
  - The same buffer needs to accessed by a routine that handles sensor data packing/transfer operations and the routine related to memory card operations
  - These routines are most likely in different .c – files
  - The variable (buffer or pointer to it) can be declared only once, but two .c-files need visibility to its definition
- The solution is to declare the variable in the module handling memory card operations (logical, as this fifo is related to memory card operations), while telling to sensor routines that the variable is extern

File: sdcard.h

```
extern fifo_t SDCard_fifo_instance;
```

File: sdcard.c

```
...
//The FIFO instance
fifo_t SDCard_fifo_instance;
...
```

File: main.c

```
#include "sdcard.h"
...
while(1) {
     ...
     if (SDCard_fifo_instance.almost_full) {
          err_code = flushSDCardFifo(SDCARD_FIFO_WM);
          ...
     }
     ...
}
```

https://www.geeksforgeeks.org/understanding-extern-keyword-in-c/

Lecture 6 – Basics of C programming language – Part 3

- Review
- Data types (cont)
- Type qualifiers
- Storage classes
- Type casting
- Pointers

C type casting

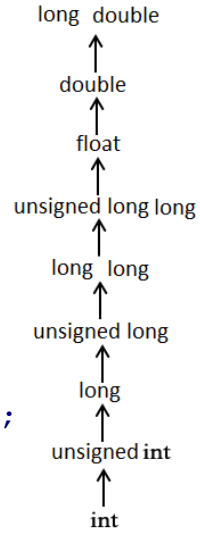
- Type casting is a way to convert a variable from one data type to another data type
- For example, if you want to store a 'long' value into a simple integer then you can type cast 'long' to 'int'
- You can convert the values from one type to another explicitly using the **cast operator**

  (type_name) expression

- Compiler does automatic conversions from smaller integer types to larger types, this is called **integer promotion**
- Casting is potentially dangerous! If you cast a "large" value to a "smaller" data type, data loss will happen
- RULE: Each cast shall feature an associated comment describing how the code ensures proper behavior across the range of possible values on the right side

```
main() {
    int sum = 17, count = 5;
    double mean;
    mean = (double) sum / count;
    printf("Value of mean: %f\n",mean);
}
```

```
main() {
    int  i = 17;
    char c = 'c'; //ascii: 99
    float sum;
    sum = i + c;
    printf("Value of sum: %f\n",sum);
}
```



Lecture 6 – Basics of C programming language – Part 3

- Review
- Data types (cont)
- Type qualifiers
- Storage classes
- Type casting
- Pointers

Declaring pointers

- Each variable in your program is stored in the memory – in other words, it has a memory location (an address)

- What is a pointer?
  - A **pointer** is a variable whose **value** is the **address** of another variable, i.e., address of a memory location
  - A pointers value can be also an address of a function – then it is called a **function pointer**
  - Like any variable or constant, you must declare a pointer before using it to store any variable or function address
- After declaration, pointer variable is uninitialized (=> can point anywhere!)

Pointer variable declaration:

```
data_type * name_p;

int * i_p;          // pointer to an integer
double * d_p; /* pointer to a double
float * f_p;       /* pointer to a float */
char * char_p;  /* pointer to a character */
my_struct_t * struct_p; /* pointer to my_struct type */


int * i_p, j_p; //WARNING!!!
//This equals:
int * i_p;
int j_p; //j_p is int variable, not a pointer to int

int *i_p, *j_p; //correct

//Best to have own line for each:
int * i_p;
int * j_p;
```

Assigning a value to pointer

- The address of any variable (or function) can be accessed via **&** -operator (address-of)

- This address can be stored to a pointer variable

```c
#include <stdio.h>
int main () {
   int  var1;
   char var2[10];
   printf("Address of var1: %x\n",&var1);
   printf("Address of var2: %x\n",&var2);
   return 0;
}
```

```
Address of var1: 2112cfc
Address of var2: 2112d00
```

Using (=dereferencing) pointers

- The variable address &var is stored to a pointer variable i_p
- And the value of the variable the pointer is pointing to, can be read with * – this is called **dereferencing**

```c
int main () {
   int  var = 20;   /* actual variable declaration */
   int  *i_p;       /* pointer variable declaration */

   i_p = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in i_p variable: %x\n", i_p );

   /* access the value using the pointer */
   printf("Value of *i_p variable: %d\n", *i_p );

   return 0;
}
```

```
Address of var variable: f46ed3dc
Address stored in i_p variable: f46ed3dc
Value of *i_p variable: 20
```

Pointers with structs – an example

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

typedef struct {
   char title[50];
   int book_id;
} Book_t;

#define NUMBER_OF_BOOKS 3

int main()
{
    Book_t library[NUMBER_OF_BOOKS]; //create array of books
    Book_t * my_ptr;       //Pointer declaration operator (points to Book_t type)
    my_ptr = &library[0];  // Address-of operator picks the
                           // address of library and assigns it to my_ptr

    //Let's check what's the size of Book_t
    printf("Size of Book_t:\t%d bytes.\n\n",sizeof(Book_t));

    //populate the library with books
    const char title_base[] = "Name of the book ";
    char letter[2] = {'A','\0'};
    for (uint8_t i=0; i<NUMBER_OF_BOOKS; i++) {

        strcpy(&my_ptr->title[0], &title_base[0]);
        strcat(&my_ptr->title[0], &letter[0]);

        my_ptr->book_id = 1000+i;
        letter[0]++; //pick the next letter
        my_ptr++; //increment the pointer to next "book slot" in the library
     }

    //Run inventory for the library

       //roll back the pointer to first book in library
       my_ptr = &library[0];

       printf("Contents of the library:\n\n");
       printf("Book Title:\t\tBook id:\n");
       for (uint8_t i=0; i<NUMBER_OF_BOOKS; i++) {
           printf("%s\t%d\n",my_ptr->title,my_ptr->book_id);
           my_ptr++; //increment the pointer
                     // to next "book slot" in the library
        }

       return 0;

}
```

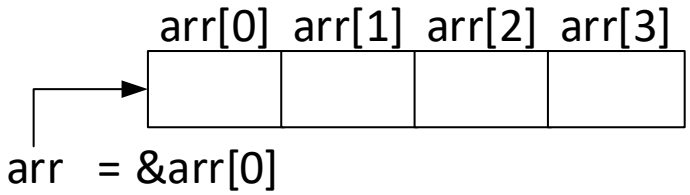
```
Size of Book_t: 56 bytes.

Contents of the library:

Book Title:             Book id:
Name of the book A      1000
Name of the book B      1001
Name of the book C      1002
```

Recap: Relation between arrays and pointers

- C does not have array variables (or string variables, which are char arrays)
- In fact, the array notation [] is actually an alternate way of using pointers
- Consider an array: int arr[4]
- Name of the array always points to the first element [0] of array
- This means that arr is equivalent to &arr[0]
- Futhermore, statement
  arr[2] =5; is identical to
  *(arr+2) = 5;
- So arr is already a pointer!



Note: strcpy and strcat are functions that take pointer arguments

```
strcpy(&my_ptr->title[0], &title_base[0]);
strcat(&my_ptr->title[0], &letter[0]);
```

my_prt->title is an character **array**, so are *title_base* and *letter* – they are already pointers

Therefore, we don't need &-operator

```
strcpy(my_ptr->title, title_base);
strcat(my_ptr->title, letter);
```