System SW

Lecture 2 – Basics of C programming language – Part 2

Jarno Tuominen / Reviewed by Sanna Määttä

Lecture 2 – Basics of C programming language – Part 2

- Review
- Data types in C

Review of last lecture

- Basic syntax
- Variables
- Constants

Review: Basic syntax

```
/*
* hello.c
* Created on: 27.8.2018
* Author: jmtuom
*/

#include <stdio.h>

int main(void) {
    printf("Hello DTEK2041 class!\n"); //This outputs formatted text
    return 0;
}
```

Basic I/O facilities like printf() is defined in stdio library, so we include it here

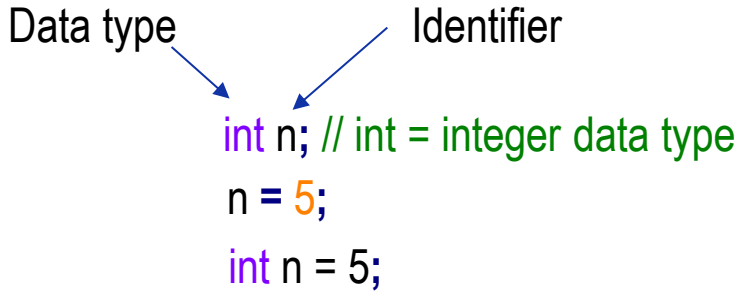
A C program must have one and only one main()-function, which is the entry point

Return integer, as ”promised” in function header

This function returns a value of type int , and it takes no arguments (void)

Review: Variables

- Must be declared before use
- Variables are uninitialized – init happens via assignment operator ”=”
- Can also be initialized at declaration
- Local or global scope



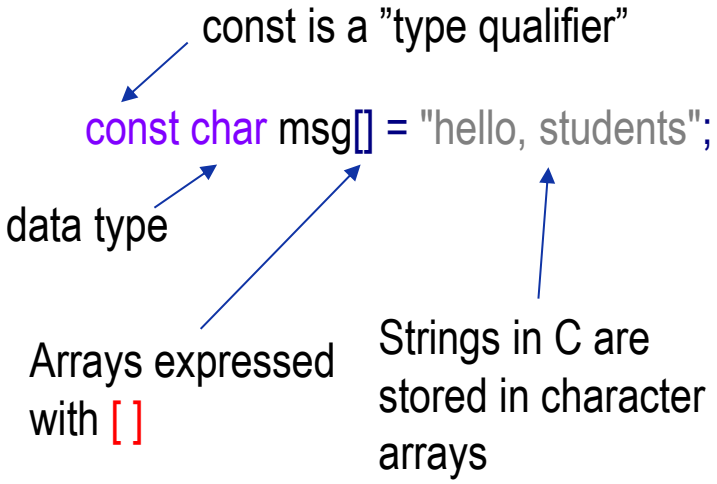

```
#include <stdio.h>
int a=12,b=22;

int main() {
        printf("Global vars can be accessed anywhere in the program %d %d",a,b);
        fun();
}

void fun() {
        int m=111,n=222;
        printf("Local vars can only be accessed inside this fun function %d %d",m,n);
}
```

Global variables

Local variables

Review: Constants

- Like variables, but their value cannot be changed during code execution
- Also called **literals**
- Constants are stored in code memory space (ROM) instead of data memory space (RAM)
- **const** keyword: qualifies variable as constant
- **char**: a data type containing a single character (1 byte)
- Here we created a constant array of characters – a string constant

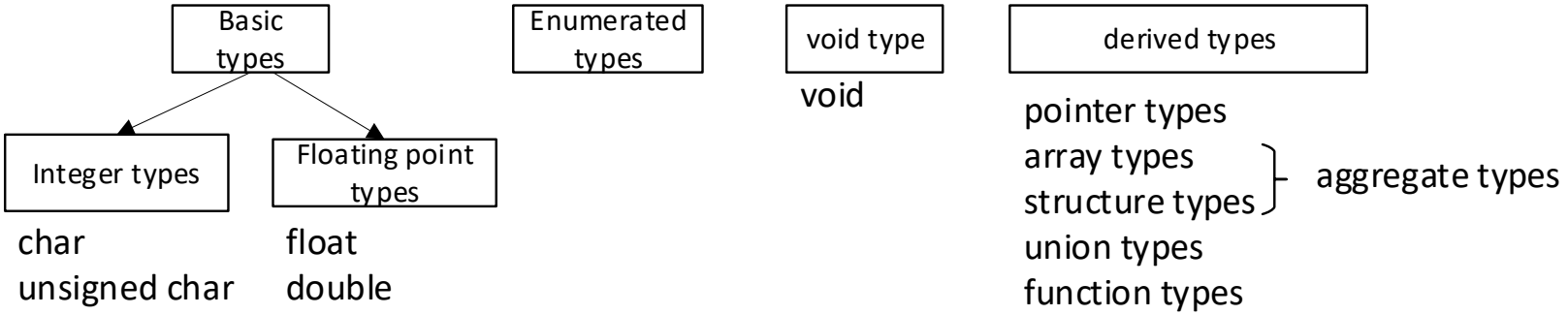


auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Lecture 2 – Basics of C programming language – Part 2

- Review
- Data types in C

The data types in C

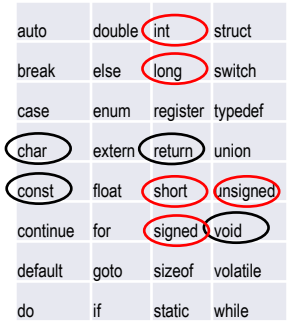


- Data types in C refer to an extensive system used for declaring variables or functions of different types
- The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

Integer data types

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

- Note: Size of int depends on the platform!
- **char** and **int** are basic data types, which can be modified with qualifiers **signed**, **unsigned**, **short** & **long**
- According to ANSI C, a shorthand for **”unsigned int”** is **”unsigned”**, which is considered *bad practice!*
- Similarly,, a shorthand for **”signed int”** is **”signed”**, *bad practice as well!*



Fixed width integer data types – with known storage size

- The platform-dependent storage size (and range) of int is confusing
- Solution: forget all the integer data types on a previous page and use **fixed width** integer types instead. #include <stdint.h> , it defines:

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

No more ambiquous int, unsigned int and char!

RULE: do not use keywords **short** and **long**
RULE: use **char** only when declaring and operating with strings

http://pubs.opengroup.org/onlinepubs/009695399/basedefs/stdint.h.html
https://barrgroup.com/Embedded-Systems/How-To/C-Fixed-Width-Integers-C99

Floating-point data types

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

- Used to store real numbers either in single precision (float) or double precision (double) floating point format, or even “double double” precision

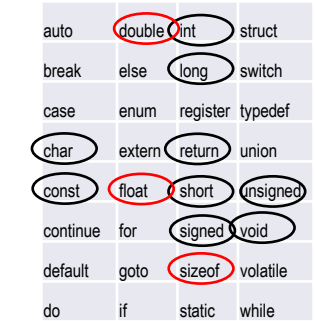
- TIP: The header file float.h defines **macros** that allow you to use these values and other details about the binary representation of real numbers in your programs

```c
#include <stdio.h>
#include <float.h>
int main() {
   printf("Storage size for float : %d \n", sizeof(float));
   printf("Minimum float positive value: %E\n", FLT_MIN );
   printf("Maximum float positive value: %E\n", FLT_MAX );
   printf("Precision value: %d\n", FLT_DIG );
   return 0;
}
```

Use ”sizeof” keyword to get storage size of any variable

macros

```
Storage size for float : 4
Minimum float positive value: 1.175494E-38
Maximum float positive value: 3.402823E+38
Precision value: 6
```



Void type

- void = “no value is available”
- Generally used for mentioning function return type when function has nothing to return, or it has no arguments

```
void main(void) {
```

- TIP/TRICK:
  - Sometimes you might have introduced variables which are not used (yet, as your are in the middle of coding work).
  - Compiler will give (harmless) warnings about unused variables, but it is a good practice to keep warning count at zero.
  - You can write a **macro** which will **cast** your variable to void type. Now your variable is ”used” and the warning is gone.

```
#define NOT_USED(X)  ((void)(X))
```

```
void main(){
  int num1 = 5; //variable, which is never used
  ...
  NOT_USED(num1); //suppress compiler warnings
 //or
 (void)(num1);
}
```

Derived data types

- Data types, that combine basic data types to create more complex data structures
  - Enumerated types (keyword: enum)
  - Structure types (keyword: struct)
  - Union types (keyword: union)
  - Array types (syntax: var[])
  - Pointer types (syntax: *my_ptr)

- Let’s have a closer look at each of those

Enumeration

Compiler treats arithmetic types like integers.

We define also a new data type. More later...

```
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;
```

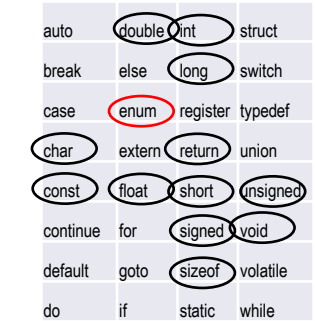
Integer value of the first member (optional as 0 is the default)

The rest get values automatically

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

```
state_t state_next = STATE_INIT; // we had typedef, no need to use enum in
                                 // front of variable name
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



Enum vs macros

```
#define   STATE_INIT 0
#define   STATE_SLEEP 1
#define   STATE_IDLE 2
#define   STATE_STOP_LOGGING 7
#define   STATE_ERROR 8
#define   STATE_POWER_OFF 9

int main()
{
  int state_next  = STATE_INIT;
  switch (state_next) {
    case STATE_ERROR:
      break;
    default:
      printf("state is: %d\n",state_next);
      break;
  }
  return 0;
}
```

- We can also use (C-preprocessor) macros/directives to define named constants
- enums follow scope rules – macros do not
- enum variables get values assigned automatically

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;
}
```

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Structure tags

- If a structure tag is left out, the struct type will be anonymous and you cannot use it for other variables
  - You need to declare the variable(s) right after the struct definition
  - You cannot declare any new variables based on this struct (as it has no tag to which to refer)

```
struct { //anonymous struct type
    char title[50];
    int book_id;
} book1, book2; //variable(s) declaration
```

- With a structure tag, you can declare more variables of this same type
  - First variable ”book1” declared at the definition, following 2 variables in a normal way
  - Note: You must still use keyword **struct** when declaring variables

```
struct Book_t { //named struct type
    char title[50];
    int book_id;
} book1;
struct Book_t book2, book3;
```

Type defined structures

- To avoid repeating ”struct” before data type (Book_t), you can create a new type definition using **`typedef`**
- The generic syntax of typedef:

```
typedef type declaration;
```

Type defined struct:

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

Book_t book1, book2;
```

Declaration (name) of the new type

Declaration of variables based on the new type

RULE: all new structures, unions and enumerations shall be named via typedef

RULE: the names of new types shall end with **_t**
(Barr: C coding standard 2018)

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

https://stackoverflow.com/questions/612328/difference-between-struct-and-typedef-struct-in-c

Typedef vs #define

- `#define` is a C-directive which is also used to define the aliases for various data types similar to **typedef** but with the following differences:
  - **typedef** is limited to giving symbolic names to types only
  - #define can be used to define alias for values as well, you can define 1 as ONE etc
  - **typedef** interpretation is performed by the compiler
  - #define statements are processed by the pre-processor

```c
#include <stdio.h>

#define TRUE  1
#define FALSE 0

int main( ) {
   printf( "Value of TRUE : %d\n", TRUE);
   printf( "Value of FALSE : %d\n", FALSE);

   return 0;
}
```

```
Value of TRUE : 1
Value of FALSE : 0
```

Structure packing

- Storage for the basic C datatypes on an x86 or ARM processor doesn't normally start at arbitrary byte addresses in memory
  - Each type except char has an *alignment requirement*
    - chars can start on any byte address
    - 2-byte shorts must start on an even address
    - 4-byte ints or floats must start on an address divisible by 4
    - 8-byte longs or doubles must start on an address divisible by 8
    - 8-byte pointers (in 64-bit architecture) must start on an address divisible by 8
- Basic C types (incl. pointers) on x86 and ARM are *self-aligned*
  - A struct instance will have the alignment of its **widest** scalar member
  - Compiler does this alignment to ensure fast access to members
  - The address of struct (in the memory) is the same as the address of its first member (no leading padding)
- 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!

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

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
```

Bit fields

```
typedef struct {
  unsigned int d;
  unsigned int m;
  unsigned int y;
} date_t;
```

- In C, we can specify the size (in bits) of structure and union members for more efficient memory usage

```
typedef struct {
  unsigned int d:5; //range: 0-31
  unsigned int m:4; //range: 0-15
  unsigned int  y:11; //range: 0-2047
} date_small_t;
```

Storage size of one int (4 bytes) is used up to 32 bits. The 33th bit would cause size to increase to 8 bytes.

```
void main() {
  printf ("Size of date_t is %d bytes\n", sizeof(date_t));
  date_t dt = {31, 12, 2018};
  printf ("Date is %d/%d/%d\n", dt.d, dt.m, dt.y);

  printf ("Size of date_small_t is %d bytes\n", sizeof(date_small_t));
  date_small_t dt2 = {31, 12, 2018};
  printf ("Date is %d/%d/%d\n", dt2.d, dt2.m, dt2.y);
}
```

```
Size of date_t is 12 bytes
Date is 31/12/2018
Size of date_small_t is 4 bytes
Date is 31/12/2018
```

Bit fields (cont.)

- stdbool.h (C99) defines a boolean data type (which can have values true and false), but the actual data type is char (uint8_t), thus allocating 1 byte each
- Using bit fields lead to huge memory savings!

```
#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

```
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));
}
```

Pointer to a structure

- You can create a **pointer** to a structure
- Pointer is a special variable type, which holds the **address** of another memory location
- When accessing a structure via pointer, member access happens via arrow ”**->**” –operator
- More about pointers later...

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

int main() {
    Book_t book;
    Book_t *book_ptr = NULL;
    book_ptr = &book;

    strcpy(book_ptr->title, "C Programming");
    book_ptr->book_id = 6495407;

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

Declare variable ”book” of type Book_t

Declare pointer to variable of type Book_t

Assign the address of ”book” to a pointer ”book_ptr”

Access members with ”->”

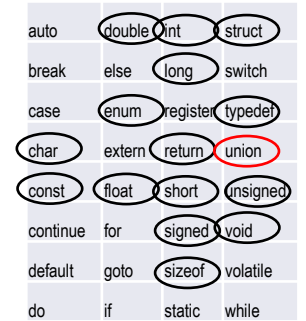
”.” works as well, as long as you dereference first

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;
```



Union - misuse

- Accessing members just like in struct – but remember, only one member can contain a value at any given time
- So, not like this

```c
int main() {

   data_t data;
   printf ("Memory size occupied by data : %d\n", sizeof (data));
   data.i = 10;                                // this gets corrupted!
   data.f = 220.5;                  // this gets corrupted!
   strcpy (data.str, "C Programming");   // this will be just fine

   printf ("data.i : %d\n", data.i);
   printf ("data.f : %f\n", data.f);
   printf ("data.str : %s\n", data.str);
   return 0;
}
```

```
Memory size occupied by data : 20
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Union – correct use

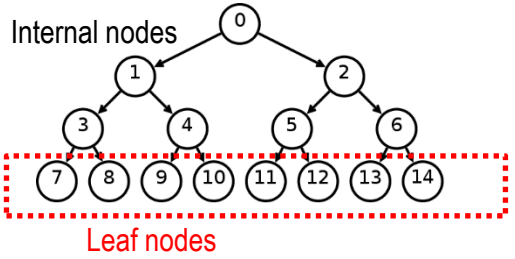
- As the union is a single memory location, you must ”use” the data value before overwriting it with the next value

```c
int main() {

  data_t data;

  data.i = 10;
  printf ("data.i : %d\n", data.i);

  data.f = 220.5;
  printf ("data.f : %f\n", data.f);

  strcpy (data.str, "C Programming");
  printf ("data.str : %s\n", data.str);

  return 0;
}
```

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

Applications of union

- Suppose we want to implement a binary tree data structure where
  - each internal node has pointers to other nodes, but no data
  - each leaf node has a double data value
- With struct, we would always reserve 16 bytes of memory for a node
- A struct containing a union with 2 members (and a flag indicating if it is a leaf node or not), will occupy only 9 bytes.




```
struct NODE {
  struct NODE *left; //4 bytes
  struct NODE *right; //4 bytes
  double data; // //8 bytes
};
```

```
struct NODE
{
    bool is_leaf; //1 byte
    union {
        struct { // 8 bytes
            struct NODE *left;
            struct NODE *right;
        } internal;
        double data; // 8 bytes
    } info;
};
```