

C

Lectures



Content

- Short Introduction
- Development environment and compilers
- C syntax
- Libraries
- `main()` function
- The very first C application
- Few words about gcc compiler.
- Other useful tools
- Reserved words in C
- Basic types
- Operators
- Statements (`if`, `for`, `while`, `switch`) and structural programming
- Functions and Modular programming
- Arrays
- Pointers
- Memory Management
- Structures



Short Introduction



Standard C

- Standardized in 1989 by ANSI (American National Standards Institute) known as ANSI C
- International standard (ISO) in 1990 which was adopted by ANSI and is known as **C89**
- As part of the normal evolution process the standard was updated in 1995 (**C95**) and 1999 (**C99**)
- C++ and C
 - C++ extends C to include support for Object Oriented Programming and other features that facilitate large software development projects
 - C is not strictly a subset of C++, but it is possible to write “*Clean C*” that conforms to both the C++ and C standards.



Elements of a C Program

- A C development environment includes
 - *System libraries and headers*: a set of standard libraries and their header files. For example see `/usr/include` and `glibc`.
 - *Application Source*: application source and header files
 - *Compiler*: converts source to object code for a specific platform
 - *Linker*: resolves external references and produces the executable module
- User program structure
 - there must be one main function where execution begins when the program is run. This function is called main
 - `int main (void) { ... },`
 - `int main (int argc, char *argv[]) { ... }`
 - UNIX Systems have a 3rd way to define `main()`, though it is not POSIX.1 compliant
 - `int main (int argc, char *argv[], char *envp[])`
 - additional local and external functions and variables

A Simple C Program

- *Create* example file: `try.c`
- *Compile* using `gcc`:
`gcc -o try try.c`
- `gcc -Wall try.c -o try`
- The standard C library *libc* is included automatically
- *Execute* program
`./try` or `try`
- Note, I always specify an absolute path
- Normal termination:
`void exit(int status);`
 - calls functions registered with `atexit()`
 - flush output streams
 - close all open streams
 - return status value and control to host environment
 - If some problems ->type
`C:\>set PATH=C:\MinGW\bin;%PATH%`

```
/* you generally want to
 * include stdio.h and
 * stdlib.h
 * */
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    printf("Hello World\n");
    exit(0);
}
```



Source and Header files

- Just as in C++, place related code within the same module (i.e. file).
- Header files (* .h) export interface definitions
 - function prototypes, data types, macros, inline functions and other common declarations
- Do not place source code (i.e. definitions) in the header file with a few exceptions.
 - inline'd code
 - class definitions
 - const definitions
- *C preprocessor* (c_pp) is used to insert common definitions into source files
- There are other cool things you can do with the preprocessor



Another Example C Program

`/usr/include/stdio.h`

```
/* comments */  
#ifndef _STDIO_H  
#define _STDIO_H  
  
... definitions and protoypes  
  
#endif
```

`/usr/include/stdlib.h`

```
/* prevents including file  
 * contents multiple  
 * times */  
#ifndef _STDLIB_H  
#define _STDLIB_H  
  
... definitions and protoypes  
  
#endif
```

`#include` directs the preprocessor to "include" the contents of the file at this point in the source file.

`#define` directs preprocessor to define macros.

`example.c`

```
/* this is a C-style comment  
 * You generally want to palce  
 * all file includes at start of file  
 * */  
#include <stdio.h>  
#include <stdlib.h>  
  
int  
main (int argc, char **argv)  
{  
    // this is a C++-style comment  
    // printf prototype in stdio.h  
    printf("Hello, Prog name = %s\n",  
        argv[0]);  
    exit(0);  
}
```


C Standard Header Files you may want to use

- Standard Headers you should know about:
 - `stdio.h` – file and console (also a file) IO: *perror, printf, open, close, read, write, scanf, etc.*
 - `stdlib.h` - common utility functions: *malloc, calloc, strtol, atoi, etc*
 - `string.h` - string and byte manipulation: *strlen, strcpy, strcat, memcpy, memset, etc.*
 - `ctype.h` – character types: *isalnum, isprint, isupport, tolower, etc.*
 - `errno.h` – defines *errno* used for reporting system errors
 - `math.h` – math functions: *ceil, exp, floor, sqrt, etc.*
 - `signal.h` – signal handling facility: *raise, signal, etc*
 - `stdint.h` – standard integer: *intN_t, uintN_t, etc*
 - `time.h` – time related facility: *asctime, clock, time_t, etc.*

Compilers



- Compilers supporting ANSI C
 - GCC (Used mostly in Unix/Linux environments)
 - Microsoft Visual C++ (supports C90. A few features of C99 standard) (Used in Windows environment)
 - ARM RealView
 - LCC
 - OpenWatcom (supports C89/90 and some C99 standard)
- There are lots of integrated development environments (IDEs) for C language available in different platforms (such as Linux and Windows).
 - The integration here means that the code development tools (i.e. debuggers) and compilers are integrated in same program. This makes development to be faster and easier.
- You can find some of them here:
<http://www.thefreecountry.com/compilers/cpp.shtml>
- In this course we use gcc and Eclipse installed in Linux.
 - <http://www.eclipse.org/downloads/>



The Preprocessor

- The C preprocessor permits you to define simple macros that are evaluated and expanded prior to compilation.
- Commands begin with a '#'. Abbreviated list:
 - `#define` : defines a macro
 - `#undef` : removes a macro definition
 - `#include` : insert text from file
 - `#if` : conditional based on value of expression
 - `#ifdef` : conditional based on whether macro defined
 - `#ifndef` : conditional based on whether macro is not defined
 - `#else` : alternative
 - `#elif` : conditional alternative



C Syntax



- “Syntax ... is the set of rules that define the combinations of symbols that are considered to be correctly structured programs in that language”
http://en.wikipedia.org/wiki/Syntax_%28programming_languages%29
- Basically in C language, syntax consists of numerous functions, which execute one after another and one at a time (modularity).



- File system overview
 - .c files= implementation(text)
 - .h files= declarations and function prototypes(text)
 - .lib files= all(binary)



Syntax Elements

- Pre-processor directives `#include <stdlib.h>`
- Function prototypes `void SomeFunction(int value);`
- Functions `void SomeFunction(int value) {...}`
- Data types
 - Variables `int result= 0;`
 - Arrays `int results[3] = {1,2,3};`
 - Structs, unions, and bitfields `struct Car{float speed; int doors;};`
 - Pointers `int* second= &results[1];`
 - Enumerations `enum Days{Mon=0,Tue,Fri=4};`
- Operators `+, -, *, /, >>, &&, |, ~, *, &, %, []`
- Selection statements(if, switch-case) `if(quit== true) { return 0; }`
- Iteration statements(for, while, do-while) `for (int i = 0; i < 5; i++) { val= i; }`
- Keywords `static, for, int, struct, void, typedef`



Libraries



- The core of the C language is small and simple.
- Special functionality is provided in the form of libraries of ready-made functions. This is what makes C so portable.
- Libraries provides you an access to many special abilities e.g. IO, without needing to know the low level complexity of reading and writing to hard disk.
- You can also make your own, but to do so you need to know how your operating system builds libraries.



- Libraries are files of ready-compiled code which we can merge with a C program at compilation time.
- Each library comes with a number of associated *header files* which make the functions easier to use in application.
- For example if you need to print to console window with standard `printf()` function or read from the console with `scanf()`, you need to include the next library in you application:

`#include <stdio.h> //Stands for standard IO`



- Why are these libraries not just included automatically?
- Because it would be a waste for the compiler to add on lots of code for maths functions, say, if they weren't needed.
- When library functions are used in programs, the appropriate library code is included by the compiler, making the resulting object code often much bigger.
- You can find C language library reference here:
http://www.acm.uiuc.edu/webmonkeys/book/c_guide/
- Open the above link and see the documentation of printf() function.
 - What function returns in different situations?
 - What is the difference between printf() and sprintf()?



Main Function



- Each program starts its execution from main function and each C program contains exactly one implementation of main() function
- Main function prototypes
 - `int main()`
 - `int main(int argc, char** argv)`
- `argc`
 - the number of program arguments
- `argv`
 - `argv[0]`= name of the program
 - `argv[1]` to `argv[argc-1]`= program arguments
 - `argv[argc]`= NULL pointer always
- Returns zero(0) when program successfully exits
- Returns non-zero in case of an error
- C99 introduced two standard macros for exit control:
 - Defined in `stdlib.h`
 - `EXIT_SUCCESS`
 - `EXIT_FAILURE`



C Syntax and Hello World

#include inserts another file. “.h” files are called “header” files. They contain stuff needed to interface to libraries and code in other “.c” files.

Can your program have more than one .c file?

What do the < > mean?

This is a comment. The compiler ignores this.

```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello world\n");
    return 0;
}
```

The main() function is always where your program starts running.

Blocks of code (“lexical scopes”) are marked by { ... }

Return ‘0’ from this function

Print out a message. ‘\n’ means “new line”.

The very first C application



- In this course we use two different ways to write and compile the applications:
 - gedit (Linux) or Notepad (Windows) for writing the code and command line for compiling the application
 - Eclipse for writing and compiling the application.
- We create the first application with gedit/ Notepad.



- Create a working folder to your home folder in Linux with next command:
mkdir c_codes
- Move to created folder:
cd c_codes
- Launch gedit:
gedit &
- When gedit is launched select **View->Highlight Mode->Sources->C**



- Write the next code with gedit:

```
/*include standard IO library*/  
#include <stdio.h>  
/*Include standard library for EXIT_SUCCESS and EXIT_FAILURE macros*/  
#include <stdlib.h>  
  
/*Implement the main function*/  
int main(int argc, char** argv)  
{  
    /*Function from stdio library, prints the given string to console*/  
    printf("Hello World!\n");  
  
    /*Return exit value from program. Thsi macro comes from stdlib*/  
    return EXIT_SUCCESS;  
}
```



- Save the file with next name and suffix:
helloworld.c
- Next thing to do is compile the application.
- This can be done with next command from command line:
gcc -Wall helloworld.c -o hello



- You should now find a compiled binary named **hello** from your working directory.
- Run the application with next command:
./hello
- You should see that the application prints text “Hello World!” to console.



Few words about gcc compiler



- The **GNU Compiler Collection** includes front ends for C, C++, Objective-C, Fortran, Java, Ada, and Go, as well as libraries for these languages.
- gcc compiler allows many options for how to compile a program.
- The next pages introduces you the most used ones.



Debug Information

- Compile hello.c with **-g** compile option, so that hello binary contains symbolic information that enables it to be debugged with the **gdb** debugger.

gcc -g hello.c -o hello



- Have the compiler generate many warnings about syntactically correct but questionable looking code with `-Wall` option. It is good practice to *a/ways* use this option!

`gcc -Wall hello.c -o hello`



- Generate optimized code on a Linux machine.

gcc -O hello.c -o hello

- Compile hello.c when it uses math library.

gcc hello.c -o hello -lm

- Follow strictly ANSI standard

gcc -ansi hello.c -o hello

- Follow strict ISO C standard

gcc -pedantic hello.c -o hello



- If the source code is in several files, say "file1.c" and "file2.c", then they can be compiled into an executable program named "app" using the following command:

gcc file1.c file2.c -o app

- Or...

gcc -c file1.c

gcc -c file2.c

gcc file1.o file2.o -o app



- Example using all options:

gcc -Wall -pedantic -ansi -g -lm first.c second.c -o app



Exercise

- Check the size of compiled hello binary.
- Now compile the application with `-g` option.
- Check the size of binary again. Why binary is now bigger?



Other useful tools



splint

- Splint is a tool for statically checking C programs for security vulnerabilities and coding mistakes.
- By default the tool is not installed, but you can very easily do that with next command:

sudo apt-get install splint



- You can check the syntax of your .c file with next command
splint some_file.c
- splint produces an output informing you about suspicious code. You should check all the warnings very carefully and correct them when necessary!



Exercise

- Install splint program in your ubuntu distribution.
- Run splint and pass the hello.c file to program.
- What warnings splint produces? Why?
- Should you correct the application somehow?



Reserved words in C



- There are few words you cannot use i.e. as variable names in your programs.
- The set of reserved words are used to build up the basic instructions of C.
- The next table contains all the reserved words in C.



Keyword	Description
asm	Keyword that denotes inline assembly language code.
auto	The default storage class.
break	Command that exits for, while, switch, and do...while statements unconditionally.
case	Command used within the switch statement.
char	The simplest C data type.
const	Data modifier that prevents a variable from being changed.
continue	Command that resets a for, while, or do...while statement to the next iteration.
default	Command used within the switch statement to catch any instances not specified with a case statement.



Keyword	Description
do	Looping command used in conjunction with the while statement. The loop will always execute at least once.
double	Data type that can hold double-precision floating-point values.
else	Statement signaling alternative statements to be executed when an if statement evaluates to FALSE.
enum	Data type that allows variables to be declared that accept only certain values.
extern	Data modifier indicating that a variable will be declared in another area of the program.
float	Data type used for floating-point numbers.
for	Looping command that contains initialization, incrementation, and conditional sections.



Keyword	Description
goto	Command that causes a jump to a predefined label. Use should be avoided.
if	Command used to change program flow based on a TRUE/FALSE decision.
int	Data type used to hold integer values.
long	Data type used to hold larger integer values than int.
register	Storage modifier that specifies that a variable should be stored in a register if possible. Fastest memory available.
return	Command that causes program flow to exit from the current function and return to the calling function. It can also be used to return a single value.
short	Data type used to hold integers. It isn't commonly used, and it's the same size as an int on most computers.



Keyword	Description
signed	Modifier used to signify that a variable can have both positive and negative values. See unsigned.
sizeof	Operator that returns the size of the item in bytes.
static	Modifier used to signify that the compiler should retain the variable's value.
struct	Keyword used to combine C variables of any data type into a group.
switch	Command used to change program flow in a multitude of directions. Used in conjunction with the case statement.
typedef	Modifier used to create new names for existing variable and function types.
union	Keyword used to allow multiple variables to share the same memory space



Keyword	Description
unsigned	Modifier used to signify that a variable will contain only positive values. See signed.
void	Keyword used to signify either that a function doesn't return anything or that a pointer being used is considered generic or able to point to any data type.
volatile	Modifier that signifies that a variable can be changed.
while	Looping statement that executes a section of code as long as a condition remains TRUE.



Basic Types



- C have a number of basic data types. Each have specific uses and advantages, depending on the application.
- In general, the data types can be divided into two categories; Integer types and Floating Point types.
- One can also think that *data type* represents the data storage unit, which reserves a particular amount of memory in compile time where we can store a particular type of data with particular size. For example **int** type usually reserves 4 bytes of memory.
- The next table contains the basic types and the range of values those type can hold.



Type	Minimum allowed range	Typical allowed range	Typical size in bytes
char	-127 → +127 or 0 → +255	-128 → +127 or 0 → +255	1
signed char	-127 → +127	-128 → +127	1
unsigned char	0 → +255	0 → +255	1
signed short int	-32767 → +32767	-32768 → +32767	2
unsigned short int	0 → +65535	0 → +65535	2
signed int	-32767 → +32767	-32768 → +32767 (antique systems) -2147483648 → +2147483647	2 or 4
unsigned int	0 → +65535	0 → +65535 (antique systems) 0 → +4294967295	2 or 4
signed long int	-2147483647 → +2147483647	-2147483648 → +2147483647 – 9223372036854775808 → +9223372036854775807 (64-bit systems)	4 or 8



Type	Minimum allowed range	Typical allowed range	Typical size in bytes
unsigned long int	0 → +4294967295	0 → +4294967295 0 → +18446744073709551615 (64-bit systems)	4 or 8
signed long long int	-9223372036854775807 → +9223372036854775807	-9223372036854775808 → +9223372036854775807	8
unsigned long long int	0 → +18446744073709551615	0 → +18446744073709551615	8
float	$1 \times 10^{-37} \rightarrow 1 \times 10^{37}$	$1 \times 10^{-37} \rightarrow 1 \times 10^{37}$	4
double	$1 \times 10^{-37} \rightarrow 1 \times 10^{37}$	$1 \times 10^{-308} \rightarrow 1 \times 10^{308}$	8
long double	$1 \times 10^{-37} \rightarrow 1 \times 10^{37}$	$1 \times 10^{-308} \rightarrow 1 \times 10^{308}$ $1 \times 10^{-4932} \rightarrow 1 \times 10^{4932}$ (x87 FPU systems)	8, 12, or 16



- Note the “Typical size in bytes” column.
- In some systems if you introduce an integer type variable in your application, the compiler reserves 2 bytes for that variable, where in some other system the compiler reserves 4 bytes for that variable.
- You should always keep this in mind since there might be data loss in some cases (4 bytes decreased to 2 bytes)!



Operators



- Operators are used with operands to build expressions. For example the following is an expression containing two operands and one operator.

$10 + 10$

- C contains the following operator groups
 - Arithmetic
 - Assignment
 - Logical/relational
 - Bitwise
 - Odds/Others



Arithmetic

- Used to calculate values
- +
- -
- /
- *
- % modulo
- -- Decrement (post and pre)
- ++ Increment (post and pre)



Assignment

- = Assignment
- *= Multiply
- /= Divide.
- %= Modulus.
- += add.
- -= Subtract.
- <<= left shift.
- >>= Right shift.
- &= Bitwise AND.
- ^= bitwise exclusive OR (XOR).
- |= bitwise inclusive OR.



Examples

```
int counter = 2;  
counter = counter + 1;  
//Can be reduced  
counter += 1;  
//or  
counter++;
```



Logical/Relational

- Used in conditional statements
- == Equal to
- != Not equal to
- > bigger than
- < smaller than
- >= bigger or equal
- <= smaller or equals
- && Logical AND
- || Logical OR ! Logical NOT



Examples

```
int main(int argv, char** argc)
{
    int first = 1;
    int second = 2;
    if(first == second)
    {
        printf("Same value");
    }
}
```



Bitwise

- $\&$ AND (Binary operator)
- $|$ inclusive OR
- \wedge exclusive OR
- \ll shift left
- \gg shift right



Odds/Others

- sizeof() size of objects and data types.
- strlen may also be of interest.
- & Address of (Unary operator)
- * pointer (Unary operator)
- ? Conditional expressions
- : Conditional expressions
- , Series operator



Statements (if, for, while, switch) and structural programming



- Structural programming is a systematic way of producing applications that are:
 - Easy to understand
 - Easy to maintain
 - Easy to test
- Structural programming requires distinct use of program structures.
- This should also become apparent from application code
 - Use indentation in program code
 - Does not make any difference to compiler, but makes it easier to read the code
- Condition and repeat statements are essentials in structured programming (if, for while...etc).



Loops and Conditionals

- Conditionals: if, else, else if, switch-case
- Switch-case keywords
 - case, default, break
- Loops: for, while, do-while
- For, while, do-while keywords
 - break, continue



If, else if and else

- Statement *if* evaluates given expression to true or false
- Notice that `zero(0) == false`, and `non-zero == true`



Example

```
#include <stdio.h>

int main(int args, char** argv)
{
    int first, second;
    printf("Give me two integers and I check which is bigger\n");
    scanf("%d %d",&first,&second);
    if(first > second)
    {
        printf("Your first number was bigger!");
    }
    else if(second > first)
    {
        printf("Your second number was bigger!");
    }
    else
    {
        printf("You gaved equals size numbers!");
    }
}
```



switch case

- *The switch and case statements help control complex conditional and branching operations.* The switch statement transfers control to a statement within its body.
- Use of the switch statement usually looks something like this:

```
switch ( expression )  
{  
    case constant-expression :  
        statements executed if the expression equals the  
        break;  
}
```



- The type of switch *expression* and case constant-expression *must be integral*.
- The value of each case constant-expression must be unique within the statement body.



Example

- Next example asks a name from the user, calculates the characters from the name, prints out the name and the amount of characters.



```
#include <stdio.h>

int main(int argv, char** argc)
{
    //Array of char where we store the name
    char name[50];
    int index = 0;
    int count = 0;
    printf("Give me your name, and I calculate the characters in it:\n");
    /*Allow only to read 49 characters -> 50 character will be reserved for \0 */
    scanf("%49s",name);

    //Check when we reached the end of the name
    while(name[index] != 0 && index < 50)
    {
        count++;
        index++;
    }

    if(count == 1)
    {
        printf("You have extremely short name, maybe it's wrong?\n");
    }

    printf("Your name %s contains %i characters!\n",name,count);
    return 0;
}
```



Control Characters

- As you could see from previous example printf() and scanf() function can contains special characters called *control characters*.



printf()

- The printf() function prototype is defined as:
`int printf (const char *format [, argument, ...]);`
- You can use next format for integers:
 - %d or %i -> printf("First int %d second %i",first,second);
 - %u for unsigned int -> printf("Unsigned int value %u", value);
 - %o for octals -> printf("Octal %o", value);
 - %x or %X for hexadecimal values -> printf("Hexadecimal value %x", value);



printf()

- You can use next formats for float values:
 - %f -> printf("Some float value %f", value)
 - %E printing with exponent
 - %g, %G system selects the best way to produce the output
- You can use next formats for chars and strings
 - %c - one character
 - %s - string (string has to end with '\0')
- Pointers
 - %p
- Accessory parts
 - Example -> printf("10 characters wide field with two decimals: %10.2f", someFvalue);



printf()

- **'\n'** new line -> `printf("First line\nSecond line");`
- **'\t'** tabulator
- **'\a'** sound "beep"
- **'\r'** Carriage return
- **'\\'** backslash character \ -> `printf("Backslash \\");`
- **'\''** apostrophe '
- **'\"'** quotation mark "
- **'\?'** question mark?



scanf()

- The scanf() function prototype is as follow:
`int scanf (const char *format [, address, ...]);`
- The information has to be given exactly in that format string demands i.e. if you want to read an integer type with scanf function the format HAS to be %i or %d.
- The control characters are the same as in printf() function.



scanf()

- Few special control characters:
 - %ns where n is the maximum amount of characters to be read -> `scanf("%10s", name);`
 - %[] where you define the characters that should be left out -> the reading ends when scanf() function detects the first unwanted character.
 - %[^] where after ^ character one defines those letter that should be left out from the string -> reading end to first this kind of letter.
 - %p for pointers



OK, We're Back.. What is a Function?

A **Function** is a series of instructions to run. You pass **Arguments** to a function and it returns a **Value**.

"main()" is a Function. It's only special because it always gets called first when you run your program.

Return type, or void

Function Arguments

```
#include <stdio.h>
/*The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello world\n");
    return
}
```

Calling a Function: "printf()" is just another function, like main(). It's defined for you in a "library", a collection of functions you can call from your program.

Returning a value



Pointers and memory allocation



Contents

1. Addresses and Pointers
2. Pointers to Array Elements
3. Pointers in Function References
4. Dynamic Memory Allocation

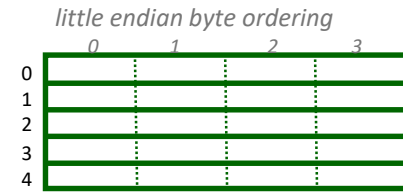


Arrays and Pointers

- A variable declared as an array represents a contiguous region of memory in which the array elements are stored.

```
int x[5]; // an array of 5 4-byte ints.
```

- All arrays begin with an index of 0



memory layout for array x

- An array identifier is equivalent to a pointer that references the first element of the array

```
- int x[5], *ptr;  
  ptr = &x[0] is equivalent to ptr = x;
```

- Pointer arithmetic and arrays:

```
- int x[5];  
  x[2] is the same as *(x + 2), the compiler will assume you mean 2  
  objects beyond element x.
```



Pointers in C

Step 1:

```
int main (int argc, argv) {
    int x = 4;
    int *y = &x;
    int *z[4] = {NULL, NULL, NULL, NULL};
    int a[4] = {1, 2, 3, 4};
    ...
}
```

Note: The compiler converts `z[1]` or `*(z+1)` to
Value at address (Address of `z` + `sizeof(int)`);

In C you would write the byte address as:

```
(char *)z + sizeof(int);
```

or letting the compiler do the work for you

```
(int *)z + 1;
```

	Program Memory	Address
x	4	0x3dc
y	0x3dc	0x3d8
	NA	0x3d4
	NA	0x3d0
$z[3]$	0	0x3cc
$z[2]$	0	0x3c8
$z[1]$	0	0x3c4
$z[0]$	0	0x3c0
$a[3]$	4	0x3bc
$a[2]$	3	0x3b8
$a[1]$	2	0x3b4
$a[0]$	1	0x3b0

82

Pointers Continued

Step 1:

```
int main (int argc, argv) {
    int    x = 4;
    int *y = &x;
    int *z[4] = {NULL, NULL, NULL, NULL};
    int  a[4] = {1, 2, 3, 4};
```

Step 2: Assign addresses to array Z

```
z[0] = a;           // same as &a[0];  
z[1] = a + 1;      // same as &a[1];  
z[2] = a + 2;      // same as &a[2];  
z[3] = a + 3;      // same as &a[3];
```

	Program Memory	Address
x	4	0x3dc
y	0x3dc	0x3d8
	NA	0x3d4
	NA	0x3d0
$z[3]$	0x3bc	0x3cc
$z[2]$	0x3b8	0x3c8
$z[1]$	0x3b4	0x3c4
$z[0]$	0x3b0	0x3c0
$a[3]$	4	0x3bc
$a[2]$	3	0x3b8
$a[1]$	2	0x3b4
$a[0]$	1	0x3b0

83

Pointers Continued

Step 1:

```
int main (int argc, argv) {
    int x = 4;
    int *y = &x;
    int *z[4] = {NULL, NULL, NULL, NULL};
    int a[4] = {1, 2, 3, 4};
```

Step 2:

```
z[0] = a;
z[1] = a + 1;
z[2] = a + 2;
z[3] = a + 3;
```

Step 3: No change in z's values

```
z[0] = (int *) ((char *)a);
z[1] = (int *) ((char *)a
                + sizeof(int));
z[2] = (int *) ((char *)a
                + 2 * sizeof(int));
z[3] = (int *) ((char *)a
                + 3 * sizeof(int));
```

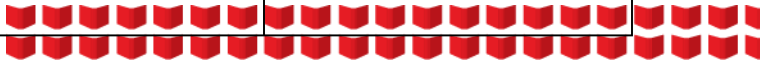
Program Memory		Address
<i>x</i>	4	0x3dc
<i>y</i>	0x3dc	0x3d8
	NA	0x3d4
	NA	0x3d0
<i>z[3]</i>	0x3bc	0x3cc
<i>z[2]</i>	0x3b8	0x3c8
<i>z[1]</i>	0x3b4	0x3c4
<i>z[0]</i>	0x3b0	0x3c0
<i>a[3]</i>	4	0x3bc
<i>a[2]</i>	3	0x3b8
<i>a[1]</i>	2	0x3b4
<i>a[0]</i>	1	0x3b0

Addresses and Pointers

- **Address:**
 - A uniquely defined memory location which is assigned to a variable.
 - A positive integer value

<An analogy with post box>

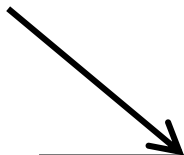
Post office box number 78	individual name Eero	contents catalog
Memory Address 66572	identifier x	contents 105



Notation for memory snapshot

Memory Address	identifier	contents
66572	x	105

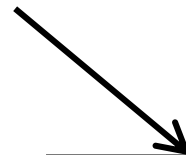
memory
address



identifier



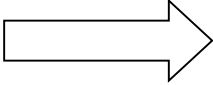
66572



x



Address operator

- To refer the address of a variable:
-  **Address operator &**
- scanf statement `scanf (“%f”, &x);`
 - > the value from the keyboard is to be stored at **&x**,
 - > the address of **x**.
- `FILE * sensor;`
- `sensor = fopen (“sensor.dat”, “r”);`
- `fscanf = (sensor, “%f %f”, &t, &motion);`



```

/*-----*/
/*  Program chapter6_1                                */
/*                                                    */
/*  This program demonstrates the relationship        */
/*  between variables and addresses.                  */
/*                                                    */

#include <stdio.h>
#include <stdlib.h>

main()
{
    /*  Declare and initialize variables.  */
    int a=1, b=2;

    /*  Print the contents and addresses of a and b.  */
    printf("a = %i; address of a = %u \n",a,&a);
    printf("b = %i; address of b = %u \n",b,&b);

    /*  Exit program.  */
    return EXIT_SUCCESS;
}
/*-----*/

```



Results of Program Chapter6_1

a 1 b 2

```
a = 1;    address of a = 65524
b = 2;    address of b = 65532
```



```

/*-----*/
/*  Program chapter6_2                                */
/*                                                    */
/*  This program demonstrates the relationship        */
/*  between variables and addresses.                  */
/*                                                    */

#include <stdio.h>
#include <stdlib.h>

main()
{
    /*  Declare variables.  */
    int a, b;

    /*  Print the contents and addresses of a and b.  */
    printf("a = %i; address of a = %u \n",a,&a);
    printf("b = %i; address of b = %u \n",b,&b);

    /*  Exit program.  */
    return EXIT_SUCCESS;
}
/*-----*/

```



Results of Program Chapter6_2

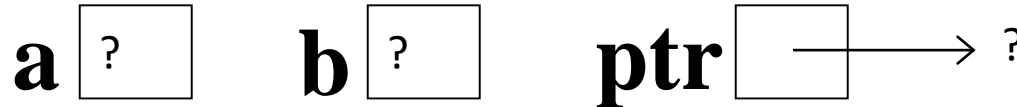
a ? b ?

```
a = 0;    address of a = 65524  
b = 150;    address of b = 65522
```



Pointer Assignment

- **Pointer**
- Special type of variable to store the address of a memory location
- **Pointer operator** * dereferencing (or indirection) operator
- `int a, b, *ptr;`



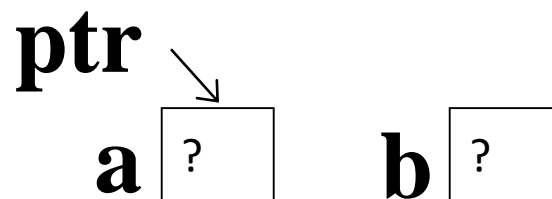
(The arrow indicates that the **ptr** is a pointer variable)



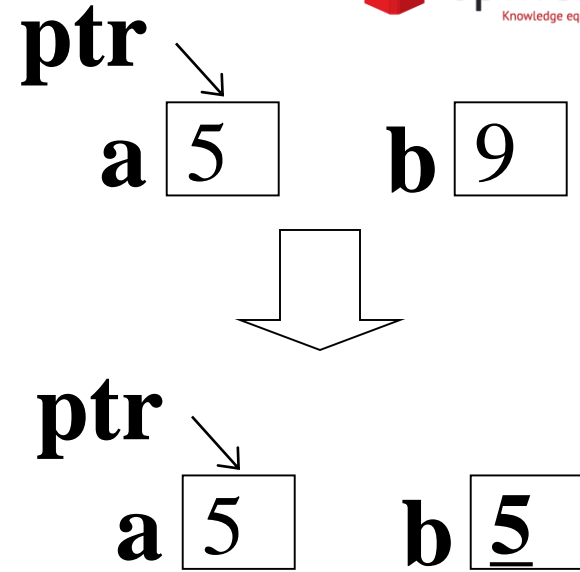
```
int a, b, *ptr ;  
ptr = &a
```

or

```
int a, b, *ptr = &a ;
```

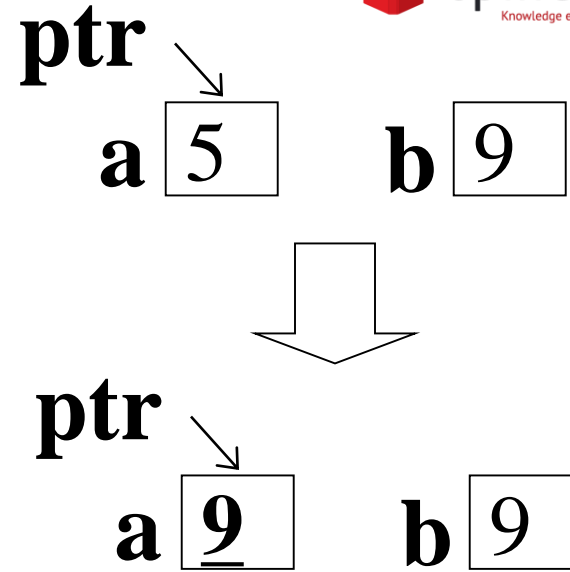


- `int a = 5, b = 9, *ptr=&a;`
- `b = *ptr;`



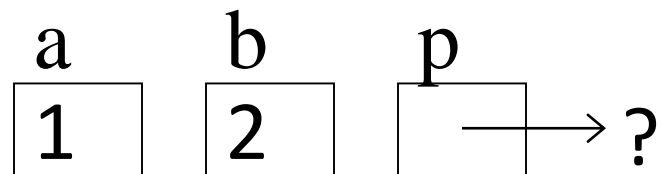
`b=*ptr` : **b** is assigned the value pointed to by **ptr**

- `int a = 5, b = 9, *ptr=&a;`
- `*ptr = b;`

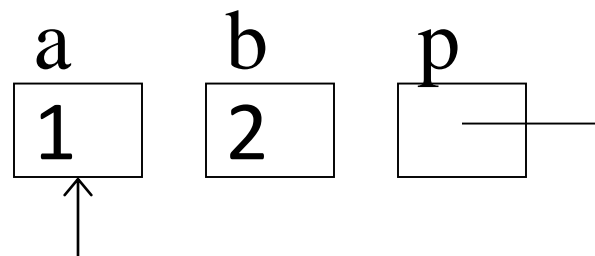


`*ptr = b;` : the value pointed to by **ptr** is assigned the value in **b**.

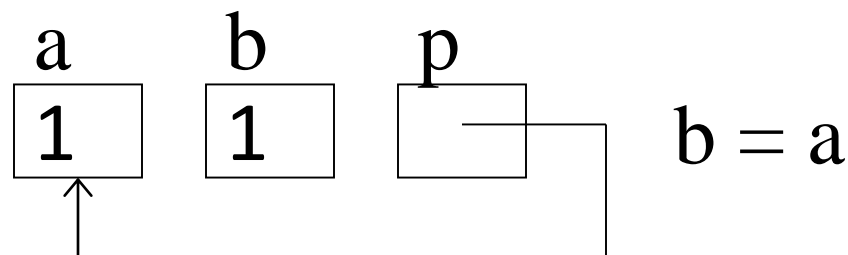
```
int a = 1, b = 2, * p ;
```



```
p = &a;
```



```
b = * p;
```



- `int i = 3, j = 5, * p = & i, * q = & j, *r;`
- `double x;`

<u>expression</u>	<u>value</u>	why?
• <code>(p == (&i))</code>	1	
• <code>* (* (&p))</code>	3	
• <code>r = (&x)</code>	illegal	
• <code>(((7*(*p)))/(*q))+7</code>	11	
• <code>(* (r = (&j))) *= (*p)</code>	15	



```

/*-----*/
/*  Program chapter6_3                                */
/*                                                    */
/*  This program demonstrates the relationship        */
/*  between variables, addresses, and pointers.      */

#include <stdio.h>
#include <stdlib.h>

main()
{
    /*  Declare and initialize variables.  */
    int a=1, b=2, *ptr=&a;

    /*  Print the variable and pointer contents.  */
    printf("a = %i; address of a = %u \n",a,&a);
    printf("b = %i; address of b = %u \n",b,&b);
    printf("ptr = %u; address of ptr = %u \n",ptr,&ptr);
    printf("ptr points to the value %i \n",*ptr);

    /*  Exit program.  */
    return EXIT_SUCCESS;
}
/*-----*/

```



Results of Program Chapter6_3

```
a = 1; address of a = 65524  
b = 2; address of b = 65522  
ptr = 65524; address of ptr = 65520  
ptr points to the value 1
```



- Address Arithmetic

1. $(\text{pointer}) = (\text{pointer})$

2. $(\text{pointer}) = (\text{pointer}) + (\text{int})$

- $(\text{pointer})++$ or $(\text{pointer})--$

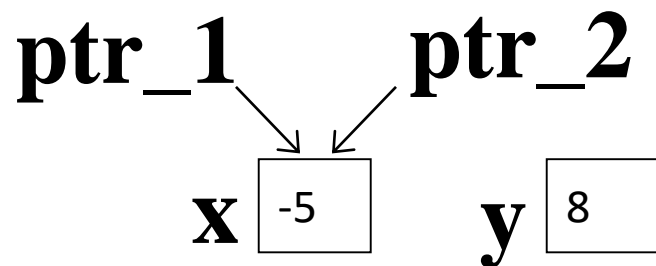
3. $(\text{int}) = (\text{pointer}) - (\text{pointer})$

4. $(\text{pointer}) = 0$ or **NULL**



Pointers to the same variable

- `int x = -5, y = 8, *ptr_1, *ptr_2 ;`
- `ptr_1 = &x;`
- `ptr_2 = ptr_1;`



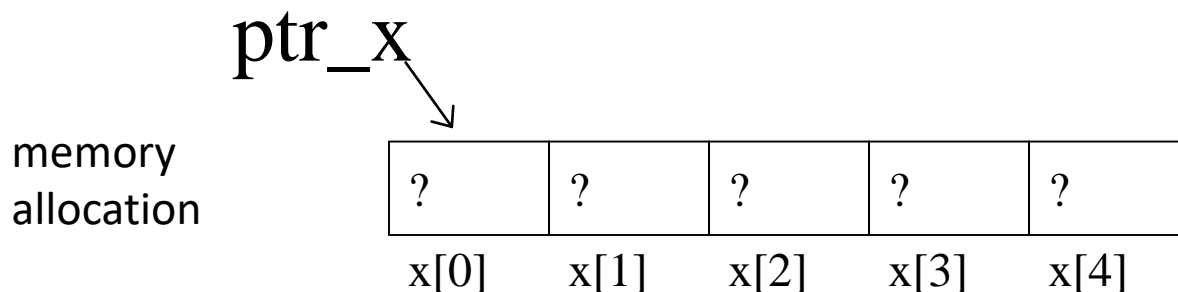
Some Common errors

- $\&y = \text{ptr_1};$ attempt to change the address of **y**
- $\text{ptr_1} = y$ attempt to change **ptr_1** to a nonaddress value
- $*\text{ptr_1} = \text{ptr_2}$ attempt to move an address to an integer variable
- $\text{ptr_1} = *\text{ptr_2}$ attempt to change **ptr_1** to a nonaddress value



- `int x[5], *ptr_x;`
- `ptr_x = &x[0];`

Pointers to Array Elements



The memory location for `x[1]` is immediately follow the memory location of `x[0]`



• Pointer Arithmetic

- depends on the machine used
- depends on the variable type

• For examples,

- Short integers (2 byte)

–
beginning : **ptr = 45530**
after **ptr++** : **ptr = 45532**

- Floating point values (4 byte)

beginning : **ptr = 50200**
after **ptr++** : **ptr = 50204**



Two-Dimensional Arrays

Array Definition:

```
int s[2][3] = { {2,4,6}, {1,5,3} };
```

Array Diagram:

2	4	6
1	5	3

Memory allocation:

		Offset
s[0][0]	2	0
s[0][1]	4	1
s[0][2]	6	2
s[1][0]	1	3
s[1][1]	5	4
s[1][2]	3	5

105

Pointers in Function References

- Function references in C,
 - Call-by-value references
 - One exception : Arrays as function parameters
- Using pointers as function parameters.
 - implements call-by-address references
 - allows to modify the values by statements within a called function



Call-by-value references

```

main ()
{
    int x = 5, y = -2;
    void switch( int a, int b);
    .....
    switch ( x, y );
    .....
}

void switch (int a, int b)
{
    int hold;
    hold = a;
    a = b;
    b = hold;
    return;
}
    
```

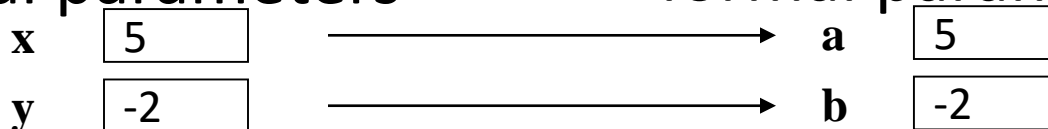
Diagram illustrating the call-by-value mechanism:

- The **Main** function contains the **Actual Parameter** (x, y).
- A **Call** is made from **Main** to the **Function** (switch).
- The **Actual Parameter** is copied into the **formal Parameter** (a, b) of the **Function**.
- The **Copy** operation is shown as a separate step, indicating that the function receives a copy of the values, not the references.

In the beginning of the function

actual parameters

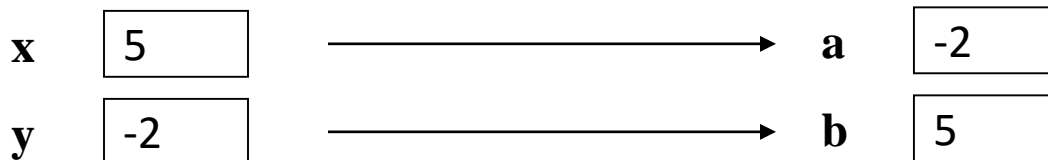
formal parameters



After the function executed

actual parameters

formal parameters

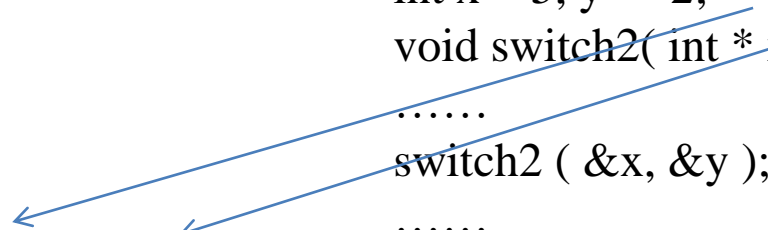


**The values have been switched in the formal parameters,
but these values are not transferred back to the actual
parameters!!!**

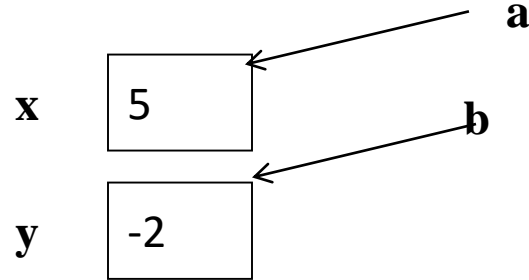


Call-by-address references

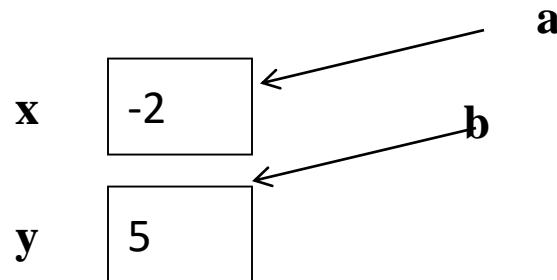
```
main ()  
{  
    int x = 5, y = -2;  
    void switch2( int * x, int * y);  
    .....  
    switch2 ( &x, &y );  
    .....  
}  
  
void switch2 (int * a, int * b)  
{  
    int hold;  
    hold = *a;  
    *a = *b;  
    *b = hold;  
    return;  
}
```



In the beginning of the function



After the function executed



Dynamic Memory Allocation

- If we do not know the exact size of the array

➔ Specify a maximum size in array definition

OR

➔ Allocate memory when a program is executed



- Dynamic memory allocation is specified using these functions:

- `void *malloc (size_t m)`

- “memory allocation”
- reserve a group of contiguous memory locations

- `void *calloc (size_t n, size_t size)`

-

- “cleared allocation”
- in addition to malloc, initialize the memory locations to a binary zero



sizeof operator :

get a size of a specific data type in byte
computes an unsigned integer (**size_t** type) value

`sizeof (int)`

`sizeof (float)`

`sizeof (double)`

`num_pts = 200;`

`void *malloc (num_pts * sizeof (int));`

`num_pts = 200;`

`void *calloc (num_pts, sizeof (int));`



void pointer

```
void *malloc ( size_t m ) ;
```

```
void *calloc ( size_t n, size_t size ) ;
```

- Both functions return a value to a pointer
- If the memory is available,
 - return the address of the memory.
- If the allocation cannot be made,
 - return NULL value.



```
/* declare variables. */  
int npts = 500;  
double *x;  
...  
/* Dynamically allocate memory */  
x = (double *)malloc(npts * sizeof (double) );
```

```
/* declare variables. */  
int npts = 500;  
double *x;  
...  
/* Dynamically allocate memory */  
x = (double *)calloc(npts, sizeof (double) );
```



- To be sure that the memory was allocated,

```
if ( x == NULL )  
{  
    printf(“Memory requested not available. \n” );  
    return EXIT_FAILURE;  
}
```



- To release allocated memory:

void free (void *ptr) ;

- you should free dynamically allocated memory when it is no longer needed.

- **/* release memory allocated to array x */
free (x) ;**



- To change the size of memory requested by **malloc** or **calloc**:

```
void *realloc ( void *ptr, size_t size) ;
```



With the use of dynamic memory allocations and releases:

- A program can be designed with a minimal amount of memory reserved.
- On system that running multiple programs, it allows more programs to run simultaneously.



Determine the maximum amount of contiguous memory

```
/*-----*/
/*  Program chapter6_5                                */
/*                                                    */
/*  This program determines the maximum contiguous    */
/*  memory allocation that can be reserved during a  */
/*  specific program execution.                        */
/*                                                    */

#include <stdio.h>
#include <stdlib.h>
#define UNIT 1000

main()
{
    /*  Declare and initialize variables.  */
    int k=1, *ptr;
```



```

/* Find maximum amount of contiguous memory */
/* available in units of thousands of integers. */
ptr = (int *)malloc(UNIT*sizeof(int));
while (ptr != NULL)
{
    free(ptr);
    k++;
    ptr = (int *)malloc(k*UNIT*sizeof(int));
}

/* Print maximum amount of memory available. */
printf("maximum contiguous memory available: \n");
printf("%i integers \n", (k-1)*UNIT);

/* Exit program. */
return EXIT_SUCCESS;
}
/*-----*/

```

Results:

```

maximum contiguous memory available:
31000 integers

```



- Lets do some exercises....



C-programming continues...

Content

1. Arrays
2. Pointers and references
3. Structs



1. Arrays

An array in C language is a collection of similar data-type, means an array can hold value of a particular data type for which it has been declared. Arrays can be created from any of the C data-types.

When we declare array, it allocates contiguous memory location for storing values. We point to a memory location with an array index value from **0** to **array size -1** in **brackets**.

In example we have an array of char type, which size is 5.

Address	00000000	0101110	01010001	00101001	00011100
index	[0]	[1]	[2]	[3]	[4]



1. Arrays

It is also possible to have more than one dimension in an array. In that case it can still hold elements of only one data type.

In below example we have a 2D char array which can hold 2x3 elements. Internally it is a one dimensional array which elements consists of arrays of size 3.

Visually, we can “see” 2 rows and 3 columns.

Address	00000000	01011110	01010001	00101001	00011100	00010110
index	[0][0]	[0][1]	[0][2]	[1][0]	[1][1]	[1][2]

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]

2 rows and 3
columns



1. Arrays

Declaring an array

Syntax

```
type array1 [size];
```

```
type array2 [row] [column];
```

Example

```
char characters[5];
```

```
int numbers [10] [5];
```

Semantics

In first row, we have declared a one dimensional array, which allocates **size** contiguous memory locations.

In second row, we have declared a 2D array, which allocates **row*column** contiguous memory locations.

In first row, we have declared an char array holding **5** elements.

In second row, we have declared a 2D int array, which holds **10** rows and **5** columns.



1. Arrays

Declaring an array

- 1) `int array[3]={12,4,5};`
- 2) `int array[3]={0};`
- 3) `int array[3]={12};`
- 4) `int array[2][3]={{1,4,5},{8,2,0}};`

An array can be defined by assigning it values in curly brace.

The array gets following elements in beside definitions:

- 1) 12, 4 and 5
- 2) 0, 0 and 0
- 3) 12, 0 and 0
- 4) 1. row 1, 4 ja 5
2. row 8, 2 ja 0

1. Arrays

Going through one dimensional array

We can operate arrays by using loops

```
for( i=0; i < size; i++)  
    array[i]=i*i;  
  
for( i=0; i < size; i++)  
    printf("%i\n", array[i]);
```

In the first example we are calculating to an array, which includes **size** elements, the squares of indexes **0**, ... , **size-1**

In the second example we are printing the values from the array



1. Arrays

Going through two dimensional array

We can operate 2D arrays by using two nested loops, where the first counter variable (i) is an index for rows and the second counter variable (j) is an index for columns.

```
for( i=0; i < rows; i++)  
    for( j=0; j < columns; j++)  
        array[i][j]=i+j;  
  
for( i=0; i < rows; i++)  
{  
    for( j=0; j < columns; j++)  
        printf("%i ",array[i][j]);  
    printf("\n");  
}
```

In the first example we are calculating to a 2D array the sums of indexes

In the second example we are printing the values from the 2D array



1. Arrays

Passing an array to the function

In C the arrays are passed to functions by default by reference (see the next section). So the function handles the actual array and not the copy of the array.

```
void square(int array[ ], int size)
{
    int i;
    for(i=0;i<size;i++)
        array[i]=array[i]*array[i];
}
```

In the function header the formal array variable is declared with square brackets.

In the function we point to element values normally with indexes in square brackets

Note! Function handles now the array of calling program and squares its elements



1. Arrays

Array of Characters (C String)

```
char characters[5+1] = "Kalle";
```

Declaring an character array with 6 elements and initialize it with 5 elements of C type char.

Internally the character array looks now like this:

Address	'K'	'a'	'l'	'l'	'e'	'\0'
---------	-----	-----	-----	-----	-----	------

Note, that the compiler adds automatically the null character `"\0"`, which indicates the end of the string.

We declare the size of array in form 5+1, so we see that the maximum size for character string is 5 and one "section" is for null character.



1. Arrays

Handling of character string in library

C library header string.h declares a useful string handling functions.

Beside is few examples:

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

```
char *strcpy(char destination[ ], char source[ ])
```

```
int strcmp(char string1[ ], char string2[ ])
```

strcpy copies the C string pointed by **source** into the array pointed by **destination**. Returns the address to destination.

strcmp compares the contents of string1 and string2 and returns a value indicating their relationship:

0, if strings are equal,

1, if string1 > string2,

-1, if string1 < string2.



1. Arrays

Reading a character string from keyboard

Standard library (**stdio.h**) function **gets** reads the character string entered to command line, which can include also punctuation marks and separate words

```
#include <stdio.h>
...
char names[32+1];

printf("Enter your name(max 32 characters) \n");
gets(names);
```

gets reads characters from command line and stores them into given array

Note! Function does not check the size of the array, but it is developers responsibility. Array overflowing makes the program work unexpectedly.



2. Pointers and references

Using pointers is a central part of C programming. Programmer must definitely understand the principle of using pointers.

Every variable allocates two adjacent memory locations: first includes the address of variable (LVALUE) and second includes the value of variable (RVALUE).

Pointer is a variable, which value (RVALUE) is an address of another variable (LVALUE). Pointer points to some variable.

Pointer

Address_ptr	Address1
-------------	----------

LVALUE

RVALUE



Normal char variable, which value is 'A'

Address1	'A'
----------	-----

LVALUE

RVALUE



2. Pointers and references

Declaring and setting pointer

Pointer

Address_ptr	Address1
-------------	----------

Normal char variable, which value is 'A'

Address1	'A'
----------	-----

```
char character = 'A';  
char *character_ptr; // or pCharacter  
  
character_ptr = &character;
```

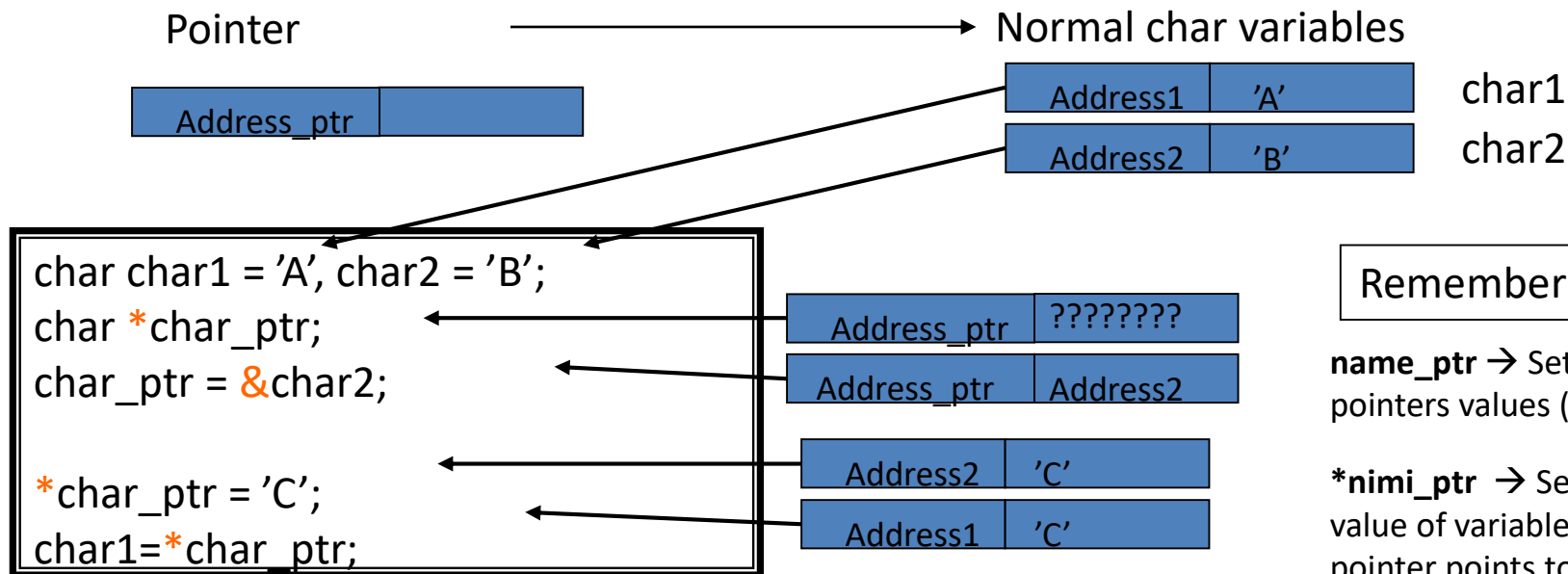
* In front of variable name → pointer

& in front of variable name → returns variable address

1. **row:** Declaring a normal char variable *character* and initializing it's value to 'A'
2. **row:** Declaring a pointer *character_ptr*, which can point to character type
3. **row:** Setting *character_ptr* to point to char variable *character*

2. Pointers and references

Writing and reading value of a variable with a pointer



Remember!

name_ptr → Set or read pointers values (addresses)

***nimi_ptr** → Set or read the value of variable to which the pointer points to



2. Pointers and references

Communication between functions

Communication between function occurs with parameters. Basically, there are two different ways to pass information to functions; **pass-by-value** and **pass-by-reference**.

Pass-by-value: The calling part of program passes the **values of parameters (RVALUE)** to function, and makes a copy of values for function parameters.

Pass-by-reference: The calling program passes the **addresses of actual parameters (LVALUE)** to function, when function can handle the values of actual parameters (RVALUE) of the calling program.



2. Pointers and references

Pass-by-reference declaration

```
int number1 = 3, number2 = 5;  
  
change(&number1,&number2);
```

```
void change(int *first, int *second)  
{  
    int temp;  
  
    temp = *first;  
    *first=*second  
    *second=temp;  
}
```

Calling program

Declaring and initializing int variables with values 3 and 5.

Calling function change(), and passing previous variables addresses to it.

In function

Pass-by-reference is declared by introducing formal parameters by pointers.

Function change() changes the variable values of calling program using local variable temp.



2. Pointers and references

Pass-by-value does not work in this case

```
int number1 = 3, number2 = 5;  
  
change(number1, number2);
```

```
void change(int first, int second)  
{  
    int temp;  
  
    temp = first;  
    first = second;  
    second = temp;  
}
```

Here is the previous page example implemented with pass-by-value. Now the variables of calling program does not change in function call, because only copies of values (RVALUE) is given to function.

In function is declared pass-by-value, when the formal parameter values get the copies of actual parameter values. Now only local variables *first* and *second* changes their values, and function does not work as expected.



2. Pointers and references

Passing an array to the function

In C the arrays are passed to the functions by default by reference. So the *-operator is not needed when declaring parameter and neither in the function.

```
void square(int array[ ], int size)
{
    int i;
    for(i=0;i<size;i++)
        array[i]=array[i]*array[i];
}
```

In the function header the formal array variable is declared with square brackets.

In the function we point to element values normally with indexes in square brackets

Note! Function handles now the array of calling part of program and squares its elements

2. Pointers and references

Passing an array to the function

In C language, the name of the array is an address of the whole array (LVALUE), so it can be the actual parameter in pass-by-reference.

```
int sum, array[5]={3,6,1,0,9};  
...  
square(array,5);  
...  
  
void square(int array[ ], int size)  
{  
    ...  
}
```

In calling program the array is passed to function by writing the name of the array for actual parameter.

After function calling the array of the calling program has changed and includes now values {9,36,1,0,81}



2. Pointers and references

Pointer arithmetic

When passing an array to the function we can use also pointer to its single elements.

Below is previous square example.

```
void square(int *first, int size)
{
    int *ptr = first;
    while(ptr-first < size)
    {
        *ptr = (*ptr) * (*ptr);
        ptr = ptr + 1;
    }
}
```

Local pointer ptr gets the value of actual parameter

Addresses are in the subsequent memory locations [first] ... [first + size]

The value of pointed variable (element of array) is handled with *-operator

Pointer arithmetic = pointer is moved from first memory location to second by increasing pointers value(address) by one.

2. Pointers and references

Pointers and strings

With pointers we can also handle character arrays (strings). The next function counts the length of the given string.

```
unsigned length(char *first)
{
    char *ptr = first;
    while(*ptr != '\0')
        ptr=ptr+1;
    return ((unsigned)(ptr-first));
}
```

Local pointer ptr gets the value of actual parameter

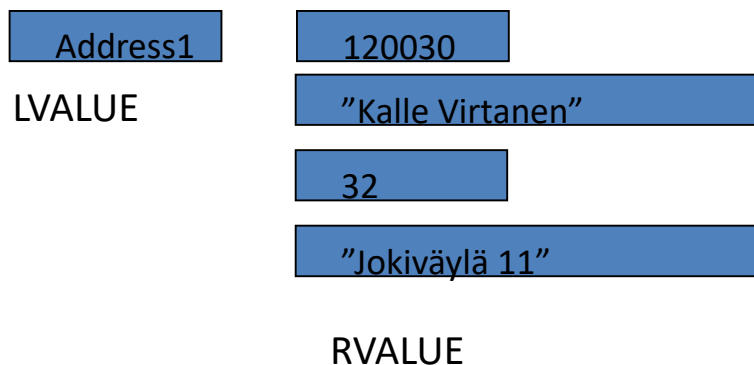
Testing, if the value of pointed variable is null character

pointer is moved from first memory location to second by increasing pointers value(address) by one.

Return the address value of last character – the address value of first character

3. Structs

C supports a structure which allows you to wrap one or more variables with different data types. Each variable in structure is called a structure member. To access struct we can use struct type variables and for structure members we can use dot operator (.)



Note!

Struct members are not necessarily in contiguous memory locations, but depends on device.



3. Structs

Struct variables can be defined in two ways

- 1) Describe the struct when introducing the variable
- 2) Define first a new global **struct type** and describe the structure. In future this struct type variables are introduced by using the type of defined struct.



3. Structs

Defining a struct variable without struct type

- 1) Describe the structure when introducing the variable

struct {*structure_member*} *variablename*;

```
int main(void)
{
    struct
    {
        int id_no;
        char name[32+1];
        unsigned age;
    } person ;

    ...
}
```

Declaration of struct variable **person**

Structure is described in the declaration. Members are introduced like a local variables, but initial values are not given.

This is a good way, if you don't need other similar struct variables.



3. Structs

Defining a struct type

- 2) Defining a new global struct type, which is used when introducing struct variable.

struct **TYPENAME** {*structure_member*};

```
struct PERSON
{
    int id;
    char * name[32+1];
};

int main(void) {
    struct PERSON dad, mom;
```

...

Declaring struct type PERSON(usually right after precompiler directives)

In function introducing struct variables dad and mom of struct type PERSON

This is good way, if you need numerous variables of the same type

Note semicolon after declaration.



3. Structs

Initializing struct in code and handling of members

```
int main(void) {  
  
    struct PERSON dad, mom;  
    mom.id=123;  
    strcpy(mom.name, "Maija");  
  
    dad ={321,"Matti"}; /* C++ */  
    ...  
}
```

To access local struct members we can use dot operator (.)

Struct members can also be initialized with curly brackets, depending on compiler(C++)



3. Structs

Copying a struct

```
int main(void) {  
  
    struct PERSON kalle1, kalle2;  
  
    kalle1.id=123;  
    strcpy(kalle1.name, "Kalle");  
  
    kalle2 = kalle1;  
}
```

The values of struct members can be copied to another struct of the same type using assignment operator (=)



3. Structs

Struct as a element of an array

```
int main(void) {  
  
    struct PERSON persons[2];  
  
    persons [0].id_no=111;  
    strcpy(persons[0].name,"Kalle");  
  
    persons [1].id_no=222;  
    strcpy(persons[1].name,"Ville");  
}
```

Defining an array **persons**, which elements are structs of type PERSON.

The struct as a element of array is used so, that the indexed element works as the name of the struct.



3. Structs

Struct as a member of a struct

```
int main(void) {  
  
    struct POINT{float x; float y;} p1;  
    struct {struct POINT cp; float radius;} circle;  
  
    p1.x = 0;  
    p1.y = 0;  
    circle.radius=5.3;  
    circle.cp.x=5;  
    circle.cp.y=2;  
}
```

In the first row we declare at same time the struct type **POINT** and variable **p1** of the same type.

In the next row we declare the struct variable **circle**, which structure includes struct variable **cp** of type **POINT** and float variable **radius**.

The members of **circle** is pointed with dot operator, in case of centre point also the inner struct members.

3. Structs

Passing a struct to function

```
int main(void) {  
  
    struct PERSON persons[2];  
    /* initializing elements of array*/  
    print (persons, 2);  
}  
  
void print (struct PERSON persons[ ], int size)  
{  
    int i;  
    for(i=0;i<size;i++)  
        printf("%s %i\n",persons[i].name, persons[i].id_no);  
}
```

Struct is passed to function by reference like other array types.

In function, structs are handled with dot operator like in calling program, where structs are created.



3. Structs

Pointer to struct

```
int main(void) {  
    struct PERSON somePerson;  
    /* initializing elements of array */  
    initStruct(&somePerson);  
    ...  
}  
  
void initStruct(struct PERSON * person)  
{  
    strcpy(person->name, "");  
    person->id_no=0;  
}
```

When struct is handled with pointer, the members of struct is handled with arrow operator -> instead of dot operator.

Here we initialize the struct to “null person”, person who has not real person details, but the members are initialized so that we can read them without crashing the program.



C Programming – File handling

- File handling in C - opening and closing.
- Reading from and writing to files.
- Special file streams stdin, stdout & stderr.
- How we SHOULD read input from the user.
- What are STRUCTURES?
- What is dynamic memory allocation?



File handling in C

- In C we use `FILE *` to represent a pointer to a file.
- `fopen` is used to open a file. It returns the special value `NULL` to indicate that it couldn't open the file.

```
FILE *fptr;  
char filename[] = "file2.dat";  
fptr = fopen (filename, "w");  
if (fptr == NULL) {  
    fprintf (stderr, "ERROR");  
    /* DO SOMETHING */  
}
```



Modes for opening files

- The second argument of `fopen` is the *mode* in which we open the file. There are three
- "r" opens a file for reading
- "w" opens a file for writing - and writes over all previous contents (deletes the file so be careful!)
- "a" opens a file for appending - writing on the end of the file



The exit() function

- Sometimes error checking means we want an "emergency exit" from a program. We want it to stop dead.
- In main we can use "return" to stop.
- In functions we can use exit to do this.
- Exit is part of the `stdlib.h` library

`exit(-1);`

in a function is exactly the same as
`return -1;`
in the main routine



Writing to a file using fprintf

- fprintf works just like printf and sprintf except that its first argument is a file pointer.

```
FILE *fptr;  
fptr= fopen ("file.dat","w");  
/* Check it's open */  
fprintf (fptr,"Hello World!\n");
```

- We could also read numbers from a file using fscanf – but there is a better way.



Reading from a file using fgets

- fgets is a better way to read from a file
- We can read into a string using fgets

```
FILE *fptr;  
char line [1000];  
/* Open file and check it is open */  
while (fgets(line,1000,fptr) != NULL) {  
    printf ("Read line %s\n",line);  
}
```

fgets takes 3 arguments, a string, a maximum number of characters to read and a file pointer. It returns NULL if there is an error (such as EOF)



Closing a file

- We can close a file simply using `fclose` and the file pointer. Here's a complete "hello files".

```
FILE *fptr;  
char filename[] = "myfile.dat";  
fptr = fopen (filename, "w");  
if (fptr == NULL) {  
    printf ("Cannot open file to write!\n");  
    exit(-1);  
}  
fprintf (fptr, "Hello World of filing!\n");  
fclose (fptr);
```



File pointer

- We use the file pointer to close the file - not the name of the file

```
FILE *fptr;  
fptr= fopen ("myfile.dat","r");  
/* Read from file */  
fclose ("myfile.dat");  
/* Ooops - that's wrong */
```



Three special streams

- Three special file streams are defined in the `stdio.h` header
- `stdin` reads input from the keyboard
- `stdout` send output to the screen
- `stderr` prints errors to an error device (usually also the screen)
- What might this do:

```
fprintf (stdout, "Hello World!\n");
```



Reading loops

- It is quite common to want to read every line in a program. The best way to do this is a while loop using fgets.

```
/* define MAXLEN at start using enum */  
FILE *fptr;  
char tline[MAXLEN]; /* A line of text */  
fptr= fopen ("sillyfile.txt","r");  
/* check it's open */  
while (fgets (tline, MAXLEN, fptr) != NULL) {  
    printf ("%s",tline); // Print it  
}  
fclose (fptr);
```



Using fgets to read from the keyboard

- fgets and stdin can be combined to get a safe way to get a line of input from the user

```
#include <stdio.h>
int main()
{
    const int MAXLEN=1000;
    char readline[MAXLEN];
    fgets (readline,MAXLEN,stdin);
    printf ("You typed %s",readline);
    return 0;
}
```



Getting numbers from strings

- Once we've got a string with a number in it (either from a file or from the user typing) we can use `atoi` or `atof` to convert it to a number
- The functions are part of `stdlib.h`

```
char numberstring[] = "3.14";  
int i;  
double pi;  
pi = atof(numberstring);  
i = atoi("12");
```

Both of these functions return 0 if they have a problem



- fgets includes the '\n' on the end
- This can be a problem - for example if in the last example we got input from the user and tried to use it to write a file:

```
FILE *fptr;  
char readfname[1000];  
fgets (readfname,1000,stdin);  
fptr= fopen (readfname,"w");  
/* oopsie - file name also has \n */
```

Even experienced programmers can make
this error



- Questions...maybe?



Questions?

