

The C Programming Language

Lecture 2

Christian A. Pagot



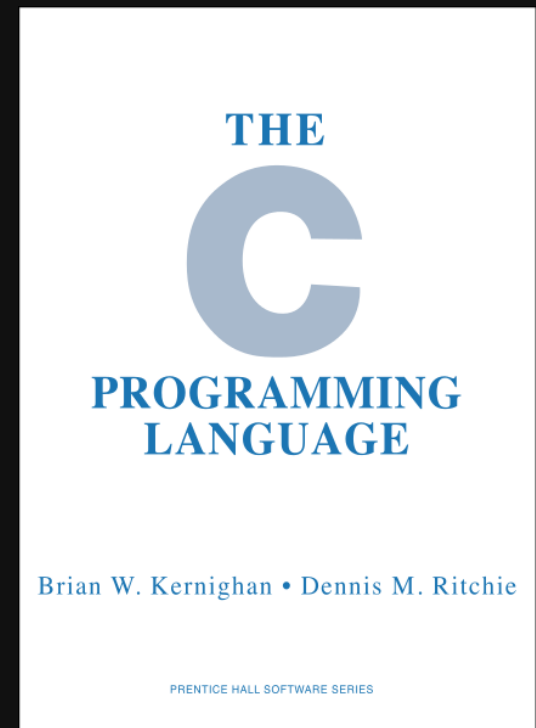
Universidade Federal da Paraíba
Centro de Informática

What is C?

C is a **imperative, general-purpose** programming language, developed in 1973 at AT&T Bell Labs by **Ken Thompson** (left) and **Dennis Ritchie** (right):



unknown



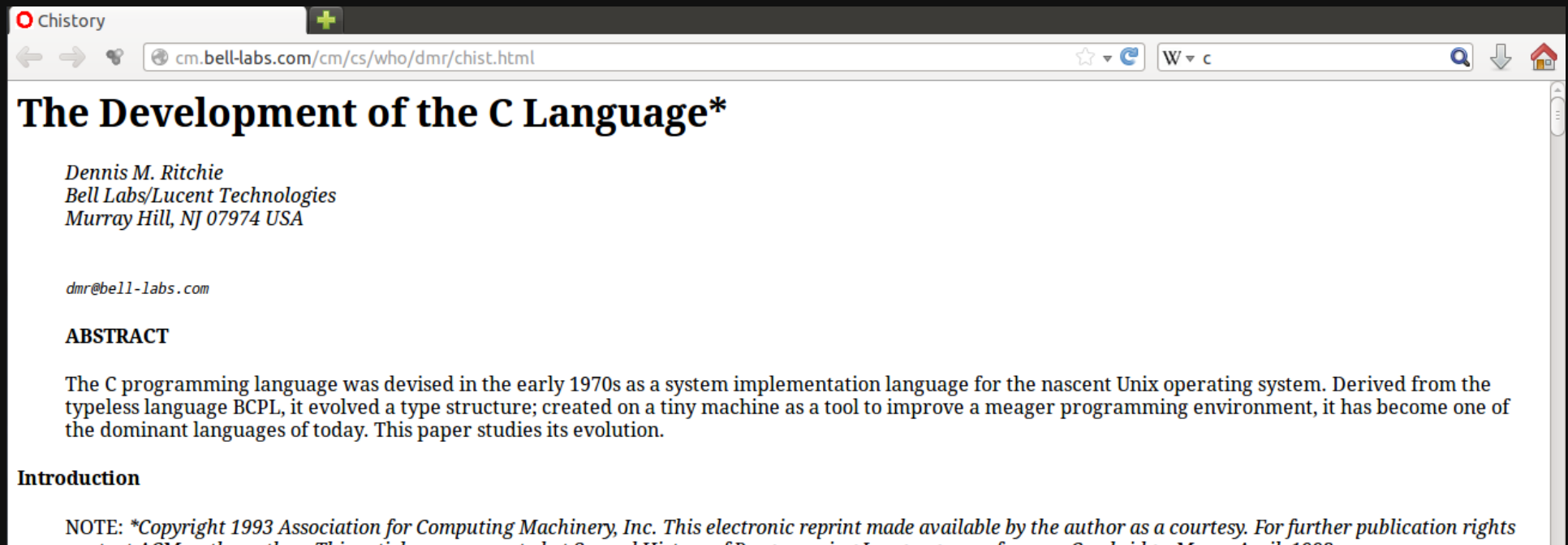
Julesmazur (Wikipedia)



The Development of the C Language

“*The Development of the C Language*”, by Dennis Ritchie:

- <http://cm.bell-labs.com/cm/cs/who/dmr/chist.html>



The screenshot shows a web browser window with the title bar "Chistory". The address bar contains the URL "cm.bell-labs.com/cm/cs/who/dmr/chist.html". The page content is titled "The Development of the C Language*" and is attributed to "Dennis M. Ritchie, Bell Labs/Lucent Technologies, Murray Hill, NJ 07974 USA". The email address "dmr@bell-labs.com" is listed. The "ABSTRACT" section states: "The C programming language was devised in the early 1970s as a system implementation language for the nascent Unix operating system. Derived from the typeless language BCPL, it evolved a type structure; created on a tiny machine as a tool to improve a meager programming environment, it has become one of the dominant languages of today. This paper studies its evolution." The "Introduction" section begins with a note: "NOTE: *Copyright 1993 Association for Computing Machinery, Inc. This electronic reprint made available by the author as a courtesy. For further publication rights contact ACM." The bottom of the page contains a footer with the logo of the Universidade Federal da Paraíba and the text "Universidade Federal da Paraíba, Centro de Informática".

The Development of the C Language*

Dennis M. Ritchie
Bell Labs/Lucent Technologies
Murray Hill, NJ 07974 USA

dmr@bell-labs.com

ABSTRACT

The C programming language was devised in the early 1970s as a system implementation language for the nascent Unix operating system. Derived from the typeless language BCPL, it evolved a type structure; created on a tiny machine as a tool to improve a meager programming environment, it has become one of the dominant languages of today. This paper studies its evolution.

Introduction

NOTE: *Copyright 1993 Association for Computing Machinery, Inc. This electronic reprint made available by the author as a courtesy. For further publication rights contact ACM.



Features of the C Language

- C is **imperative** (procedural).
 - Instructions define the actions of the processor.
- C instructions **map easily to machine instructions**.
- C is meant to be **cross-platform**.
 - Source code can be compiled to different hardware with minimal modifications.
- C uses **lexical scoping**.
 - Variable scope defined by its position in the source code.



Features of the C Language

- C uses **static** type system.
 - Types are checked during compile time.
- C supports **recursion**.
 - A function can call itself.
- In C, function parameters are **passed by value**.
 - Copies.
- C is **weakly** typed.
 - Casting.



C Compilers

- There are several **C compilers** around. Some examples are:
 - Open Watcom C/C++.
 - CodeWarrior.
 - **GCC**.
- More C compilers on:

https://en.wikipedia.org/wiki/List_of_compilers#C_compilers



Versions of C

- Since its creation, C has undergone several **improvements**.
- Resulting **versions** have been **published** as **standards**:
 - ANSI C (1990), ratified as ISO/IEC 9899:1990.
 - `gcc` option: `-ansi` or `-std=c90`
 - ISO/IEC 9899:1999.
 - `gcc` option: `-std=c99`
 - ISO/IEC 9899:2011.
 - `gcc` option: `-std=c11`

At the time of this writing, **my gcc defaults** to `-std=gnu89`, which stands for the **GNU-extended** version of the **ISO/IEC 9899:1990**.



C Program Example

The program below computes the summation of all integers within a given closed interval:

summation.c

```
#include <stdio.h>

int Sum( int begin, int end ) {
    int i;
    int acc = 0;
    for ( i = begin; i <= end; i++ )
        acc += i;

    return acc;
}

int main () {
    int a = 1;
    int b = 5;
    int sum = Sum( a, b );
    printf( "Sum: %i\n", sum );
    return 0;
}
```

Compile and run

```
~$ gcc summation.c
~$ ./a.out
```

or

```
~$ gcc summation.c -o summation
~$ ./summation
```



C Data Types

- Basic types.
- Structured types.
- User defined types.
 - They are, basically, combinations of the above data types identified by a user-defined name.



Basic Data Types

- char
- int
- float
- double
- pointers

Optional specifiers*:

- signed
- unsigned
- short
- long

*may not apply to all numeric types.

char
signed char
unsigned char
short
short int
signed short
signed short int
unsigned short
unsigned short int
int

signed int
unsigned
unsigned int
long
long int
signed long
signed long int
unsigned long
unsigned long int
long long

long long int
signed long long
signed long long int
unsigned long long
unsigned long long int
float
double
long double



Structured (Aggregate) Data Types

- C offers 2 types of structured data types:
 - Arrays.
 - Structs.



Structs

- Is an aggregate that might contain variables of **different types**.
- Given a variable of type `struct`, its variables can be accessed through the `'.'` operator.

- Example

`structs.c`

```
struct Date {  
    int day;  
    int month;  
    int year;  
};  
  
int main( void ) {  
    struct Date x;  
    x.day = 5;  
    x.month = 2;  
    x.year = 2016;  
  
    return 0;  
}
```



Arrays

*“(...) an **array data structure**, or simply an **array**, is a data structure consisting of a **collection** of elements (values or variables), each identified by at least one array **index** or key. An array is stored so that the **position** of each element **can be computed from its index tuple** by a mathematical formula.”*

Array data structure, Wikipedia



Arrays in C

- Traditionally of **fixed, static** size.
- Usually, all elements are of the **same type**.
- **Does not** carry information about its **size**!
- May be **multidimensional**.
- Example

example_26.c

```
#include <stdio.h>

int x[4] = { 10, 20, 30, 40 };

int main() {
    int i;
    for ( i = 0; i < 4; i++ )
        printf( "%i ", x[i] );

    return 0;
}
```



Multidimensional Arrays in C

- **Multidimensional** arrays can be obtained by applying the array concept **recursively**.
- **Example**

example_27.c

```
#include <stdio.h>

int x[2][3] = { { 10, 20, 30}, { 40, 50, 60 } };


int main() {
    int i, j;
    for ( i = 0; i < 2; i++ )
        for ( j = 0; j < 3; j++ )
            printf( "%i ", x[i][j] );

    return 0;
}
```

Logical representation
of the 2D array.



	0	1	2
0	10	20	30
1	40	50	60



addr + 0	10
addr + 1	20
addr + 2	30
addr + 3	40
addr + 4	50
addr + 5	60

Actual distribution of the
array elements in memory.



Another C Program Example

The program below computes the average of the following integers: 2, 2, 3, 4 and 5:

average.c

```
#include <stdio.h>

float Average( int *w, int n ) {
    int i;
    int avg;
    for ( i = 0; i < n; i++ )
        avg += w[i];

    return avg / n;
}

int main () {
    int x[5] = {2, 2, 3, 4, 5};
    int num = 5;
    float avg = Average( x, num );
    printf( "Average: %f\n", avg );
    return 0;
}
```

**Oops! We were expecting
the average to be 3.2!
How do we approach
this problem?**



Debuggers

- “(...) a computer program that is used to test and debug other programs (...).”

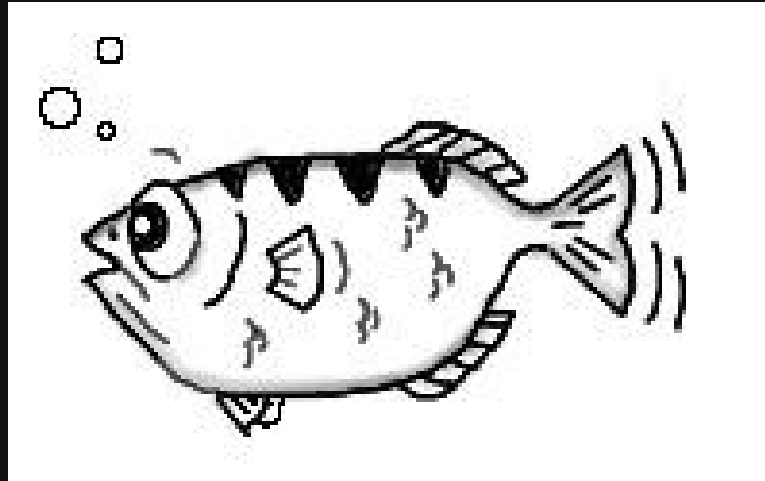
Debugger, Wikipedia.

- Examples:
 - Microsoft Visual Studio Debugger.
 - LLDB.
 - GDB.



GDB

- GDB, or the GNU Project Debugger
 - Popular debugger tool used among Unix/Linux programmers.
 - It comes, usually, pre-installed in several Linux distributions.



The Archer Fish,
the GDB mascot.



GDB

- Features (as of version 7.10):
 - Can be used to debug C and C++ programs.
 - Partial support to some other languages.
 - Text-based.
 - “Normal”, temporary and conditional breakpoints.
 - Single-stepping.
 - Resume.
 - Watchpoints.
 - Variable inspection.
 - Call stack inspection.
 - Etc.



GDB Usage Example

Back to the broken C program:

average.c

```
#include <stdio.h>

float Average( int *w, int n ) {
    int i;
    int avg;
    for ( i = 0; i < n; i++ )
        avg += w[i];

    return avg / n;
}

int main () {
    int x[5] = {2, 2, 3, 4, 5};
    int num = 5;
    float avg = Average( x, num );
    printf( "Average: %f\n", avg );
    return 0;
}
```

Recompile the program with the following command:

```
~$ gcc -g3 average.c -o average
```

Inserts debugging information into the executable.



GDB Usage Example

- After the recompilation, invoke GDB on the executable file:

```
~$ gdb ./average
```

- From within the GDB environment, let's issue the following commands:

- List the source code from within GDB:

```
(gdb) list 1, 100
```

or

```
(gdb) l 1, 100
```

- Set a breakpoint at line 13:

```
(gdb) break 13
```

or

```
(gdb) b 13
```



GDB Usage Example

- From within the GDB environment, let's issue the following commands (cont.):

- Run the program:

```
(gdb) run
```

or

```
(gdb) r
```

- Lets inspect the value of variable **x**:

```
(gdb) print x
```

or

```
(gdb) p x
```

- Execute line 14 (just one step):

```
(gdb) step
```

or

```
(gdb) s
```

- Lets inspect the value of variable **x** again:

```
(gdb) p x
```



GDB Usage Example

- From within the GDB environment, let's issue the following commands (cont.):

- Which is the next line to be executed?

```
(gdb) frame
```

or

```
(gdb) f
```

- One more step:

```
(gdb) s
```

- Step over the call to **Average()**:

```
(gdb) next
```

or

```
(gdb) n
```

Differently from
step, next
do not step
into functions!

- Inspect the value of variable **avg**:

```
(gdb) p avg
```

It seems that the problem is within
the **Average()** function!



GDB Usage Example

- From within the GDB environment, let's issue the following commands (cont.):

- Let's restart the program:

```
(gdb) r
```

- We've got stuck at line 13 again! First, let's print breakpoint information for the program:

```
(gdb) info breakpoint
```

or

```
(gdb) info b
```

- Now, delete breakpoint 1:

```
(gdb) delete 1
```

or

```
(gdb) d 1
```

- Step until we reach line 15.



GDB Usage Example

- From within the GDB environment, let's issue the following commands (cont.):

- Now, let's step into the function **Average()**:

```
(gdb) s
```

→ step steps into functions!

- Let's check the value of the local variable **avg**:

```
(gdb) p avg
```

→ **avg was not properly initialized!**

- Initialize **avg** with 0!
- Now, recompile the program (do not close GDB!).
- Rerun the program, without breakpoints, and check the answer. → **It seems that we still have a problem!**



GDB Usage Example

- From within the GDB environment, let's issue the following commands (cont.):
 - Set a breakpoint at the function **Average()** :

```
(gdb) b Average
```
 - Rerun the program (it will stop within **Average()**).
 - Set a watchpoint for when the loop finishes (**i == n**):

```
(gdb) watch i == n
```
 - Continue until next breakpoint / watchpoint.

```
(gdb) continue
```

or

```
(gdb) c
```



GDB Usage Example

- From within the GDB environment, let's issue the following commands (cont.):
 - Check the value of variables **avg** and **n**.
 - Step until we leave **Average()**.
 - Check the value that was returned by the function.
- ↘ **Damn! The value is incorrect! So what???**
- The value returned by **Average()** is the result of a integer division! We have to cast one of the operators (e.g. **(float) avg/n**) to force a float division!
 - Apply cast, recompile, delete all breakpoints and rerun!



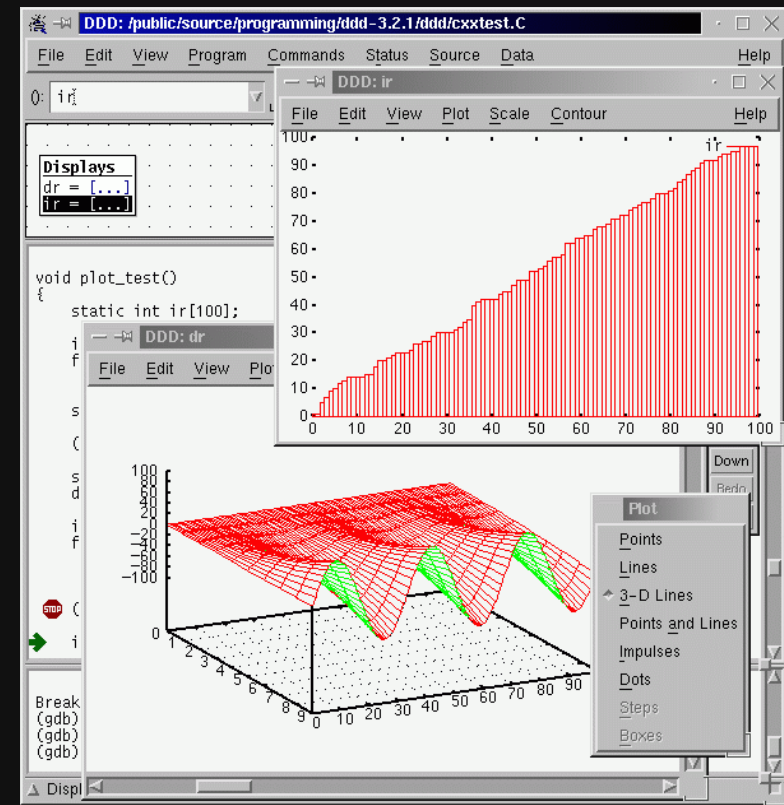
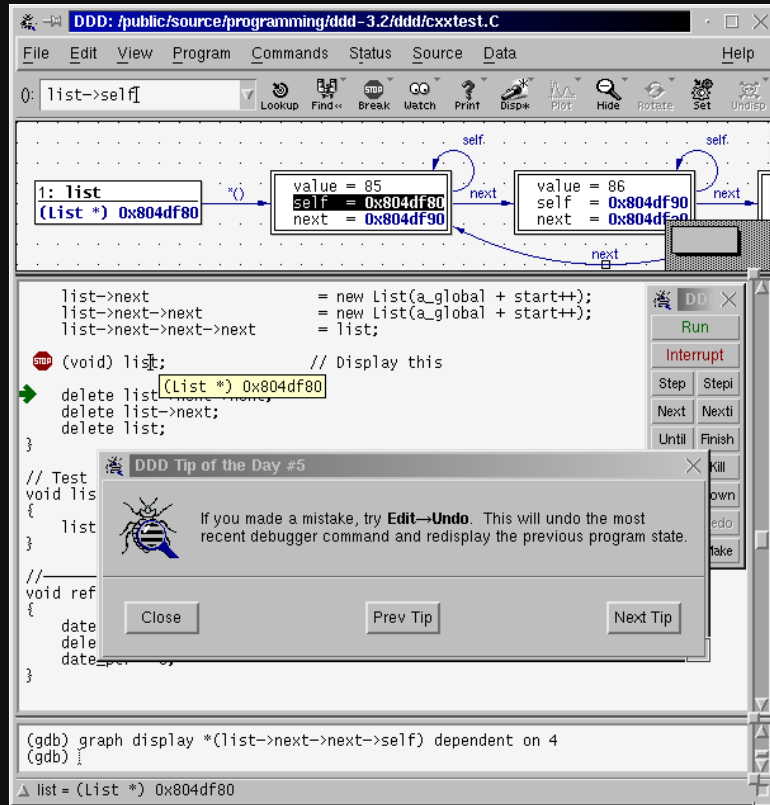
GDB Summary

- GDB is a quite powerful debugger tool.
- However, GDB does not allow one to easily follow the source code during a debug session.
- Some **GDB front ends** were developed, most notably:
 - CTRL + x + a : splits the GDB screen in command and source code windows (buggy!).
 - cgdb: curses based GDB front end.
 - Eclipse: an IDE that may use GDB as its debugging tool.
 - **DDD** : the Data Display Debugger.



The Data Display Debugger (DDD)

- It is a graphical interface to GDB.



- More on:

- <https://www.gnu.org/software/ddd>



Back to C Data Types... Pointers!

- A **pointer** variable is a **memory location** into which **data** (i.e. a **memory address**) can be **stored**.



Pointer Variable Example

Compile and run the following code from within GDB:

example_23.c

```
int main() {  
    int x;  
    int *px;  
    x = 25;  
    px = &x;  
  
    return 0;  
}
```

00000000000000000000000000011001₂

byte

x

Addr.	Value
addr1 - 1	...
addr1 + 0	...
addr1 + 1	...
addr1 + 2	...
addr1 + 3	...

addr2 + 0	...
addr2 + 1	...
addr2 + 2	...
addr2 + 3	...
addr2 + 4	...
addr2 + 5	...
addr2 + 6	...
addr2 + 7	...

(gdb) b 4 —> Set a breakpoint at line 4.
(gdb) r —> Run (stops at line 4).
(gdb) p &x —> Print the address of x.
(gdb) p x —> Print the integer value stored at x.
(gdb) s



Pointer Variable Example

Compile and run the following code from within GDB:

example_23.c

```
int main() {
    int x;
    int *px;
    x = 25;
    px = &x;

    return 0;
}
```

00000000000000000000000000011001₂

00000000000000000111111111111111
111111111111111111101110100000100₂

byte

x

px

Addr.	Value
addr1 - 1	00011001
addr1 + 0	00000000
addr1 + 1	00000000
addr1 + 2	00000000
addr1 + 3	00000000

addr2 + 0	...
addr2 + 1	...
addr2 + 2	...
addr2 + 3	...
addr2 + 4	...
addr2 + 5	...
addr2 + 6	...
addr2 + 7	...

(gdb) b 4 —> Set a breakpoint at line 4.
 (gdb) r —> Run (stops at line 4).
 (gdb) p &x —> Print the address of x.
 (gdb) p x —> Print the integer value stored at x.
 (gdb) s —> Execute line 4 (x = 25).
 (gdb) p x —> Print the integer value stored at x.
 (gdb) p px —> Print the address stored at px.
 (gdb) s



Pointer Variable Example

Compile and run the following code from within GDB:

example_23.c

```
int main() {
    int x;
    int *px;
    x = 25;
    px = &x;

    return 0;
}
```

00000000000000000000000000011001₂

00000000000000000111111111111111
111111111111111111101110100000100₂

byte

x

px

Addr.	Value
addr1 - 1	00011001
addr1 + 0	00000000
addr1 + 1	00000000
addr1 + 2	00000000
addr1 + 3	00000000

addr2 + 0	00000100
addr2 + 1	11011101
addr2 + 2	11111111
addr2 + 3	11111111
addr2 + 4	11111111
addr2 + 5	01111111
addr2 + 6	00000000
addr2 + 7	00000000

```
(gdb) b 4 —> Set a breakpoint at line 4.
(gdb) r —> Run (stops at line 4).
(gdb) p &x —> Print the address of x.
(gdb) p x —> Print the integer value stored at x.
(gdb) s —> Execute line 4 (x = 25).
(gdb) p x —> Print the integer value stored at x.
(gdb) p px —> Print the address stored at px.
(gdb) s —> Execute line 5 (px = &x).
(gdb) p px —> Print the address stored at px.
```



Examining Data with GDB: `print`

- `print` is the most **common** way to **examine data**, and is based on **expression evaluation**.
- `print` is able to **format the output!**
- Back to the previous example: set a breakpoint at line 7 (`return 0`) and run.

`example_23.c`

```
int main() {  
    int x;  
    int *px;  
    x = 25;  
    px = &x;  
  
    return 0;  
}
```

Experiment `print` with these arguments:

<code>(gdb) p x</code>	→	Print the integer value stored at <code>x</code> .
<code>(gdb) p /x x</code>	→	Print the value at <code>x</code> in hexa format.
<code>(gdb) p /t x</code>	→	Print the value at <code>x</code> in binary format.



Examining Data with GDB: **x**

- The **x** command allows for **low-level** data examination.
- It prints the **contents** of **memory** positions in a specified **format**.



Examining Data with GDB: **x**

Back to the previous example:
set a breakpoint at line 7
(**return 0**) and run.

example_23.c

```
int main() {  
    int x;  
    int *px;  
    x = 25;  
    px = &x;  
  
    return 0;  
}
```

byte →

Addr.	Value
x addr1 - 1	00011001
addr1 + 0	00000000
addr1 + 1	00000000
addr1 + 2	00000000
addr1 + 3	00000000

px

addr2 + 0	00000100
addr2 + 1	11011101
addr2 + 2	11111111
addr2 + 3	11111111
addr2 + 4	11111111
addr2 + 5	01111111
addr2 + 6	00000000
addr2 + 7	00000000

Experiment **x** with these arguments:

(gdb) x &x	→	Print the value at address &x (last format).
(gdb) x/t &x	→	Print the value at address &x in binary.
(gdb) x/d &x	→	Print the value at address &x in decimal.
(gdb) x/4tb &x	→	Print 4 bytes in binary starting at addr. &x.
(gdb) x/8tb &px	→	Print 8 byt. in binary starting at addr. &px.



Pointer Arithmetic

example_24.c

```
int x[4] = { 10, 20, 30, 40 };

int main() {
    int *pint = x;

    *pint = 0;
    *(pint + 1) = 0;
    *(pint + 2) = 0;
    *(pint + 3) = 0;

    char *pbyte = ( char* ) x;

    *pbyte = 255;
    *(pbyte + 1) = 255;
    *(pbyte + 2) = 255;
    *(pbyte + 3) = 255;

    return 0;
}
```

	Addr.	Value
pint		
pbyte	addr + 0	00001010
pbyte+1	addr + 1	00000000
pbyte+2	addr + 2	00000000
pbyte+3	addr + 3	00000000
pint+1	addr + 4	00010100
	addr + 5	00000000
	addr + 6	00000000
	addr + 7	00000000
pint+2	addr + 8	00011110
	addr + 9	00000000
byte	addr + 10	00000000
	addr + 11	00000000

■ ■ ■



Pointer Arithmetic

example_24.c

```
int x[4] = { 10, 20, 30, 40 };

int main() {
    int *pint = x;

    *pint = 0;
    *(pint + 1) = 0;
    *(pint + 2) = 0;
    *(pint + 3) = 0;

    char *pbyte = ( char* ) x;

    *pbyte = 255;
    *(pbyte + 1) = 255;
    *(pbyte + 2) = 255;
    *(pbyte + 3) = 255;

    return 0;
}
```

Dereferencing
can be
equivalently
rewritten with []!

example_25.c

```
int x[4] = { 10, 20, 30, 40 };

int main() {
    int *pint = x;

    pint[0] = 0;
    pint[1] = 0;
    pint[2] = 0;
    pint[3] = 0;

    char *pbyte = ( char* ) x;

    pbyte[0] = 255;
    pbyte[1] = 255;
    pbyte[2] = 255;
    pbyte[3] = 255;

    return 0;
}
```

Sounds
familiar?



Pointers vs. Arrays

- Are arrays **pointers**?
- If there is a difference, **could you point it out?**

To fully understand **how arrays** actually **work** in C, we need to first understand how **storage classes** and **parameter passing** work!



References

- **Learning C with GDB.** Alan O' Donnell.
 - <https://www.recurse.com/blog/5-learning-c-with-gdb>

