The C Programming Language: Part I

Lecture 2

1107186 – Estruturas de Dados

Prof. Christian Azambuja Pagot
CI / UFPB

Universidade Federal da Paraíba
Centro de Informática

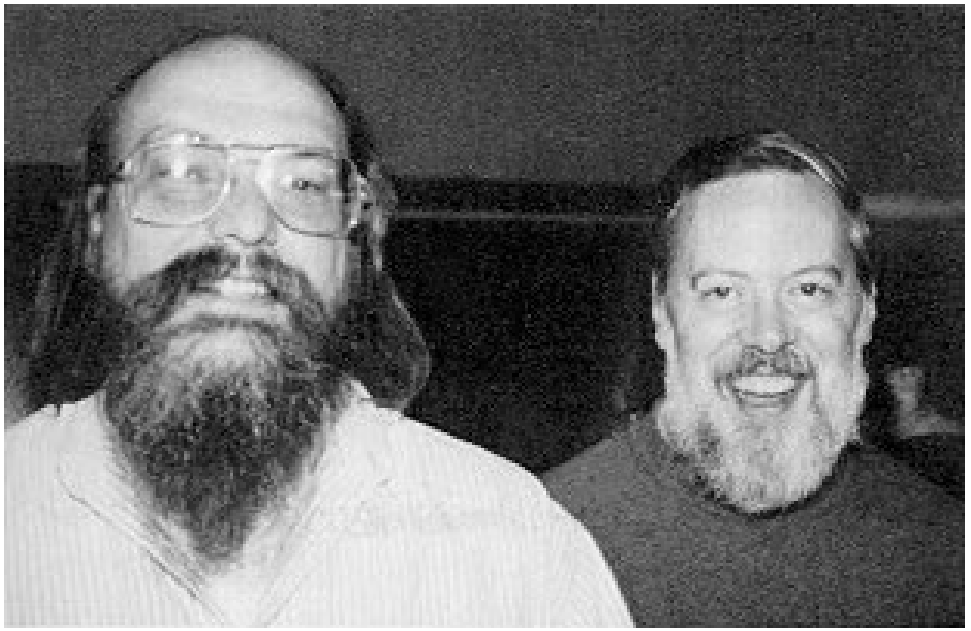
What is C?

- C is a **imperative** (procedural) **general-purpose** programming language.
- **Example**:

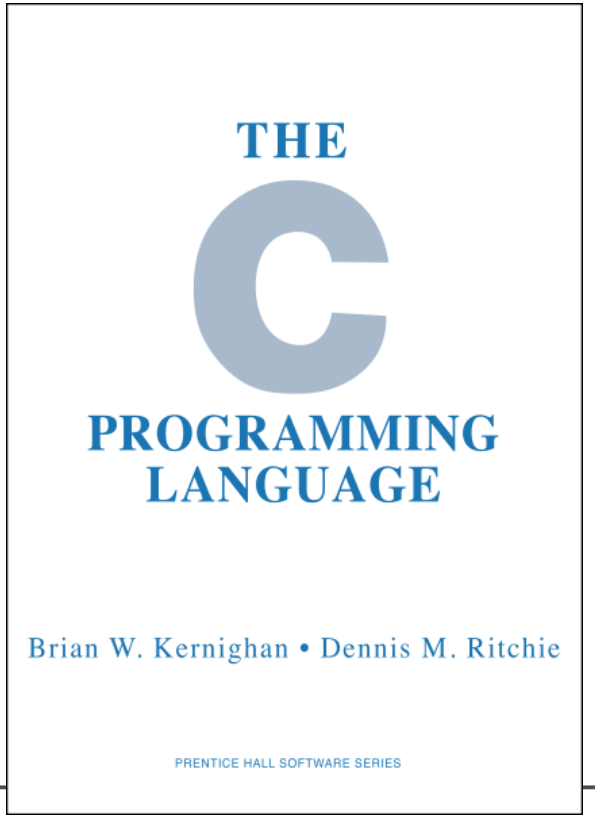
```
1 int main(void)
2 {
3     printf("hello, world\n");
4 }
5
6
7
```

Where it came from?

- C was developed in 1973 at AT&T Bell Labs by **Ken Thompson** (left) and **Dennis Ritchie** (right):



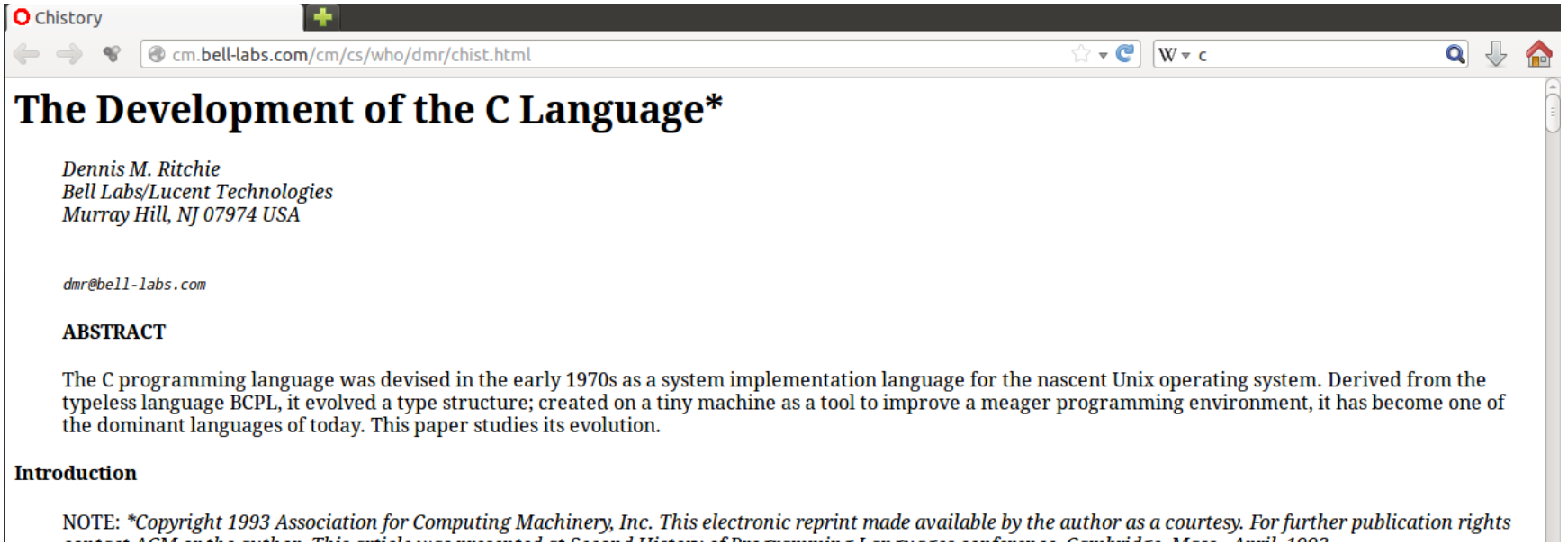
Jargon File



Julesmazur (Wikipedia.org)

Where it came from?

- *"The Development of the C Language"*, by Dennis Ritchie:
  - **http://cm.bell-labs.com/cm/cs/who/dmr/chist.html**



Chistory

cm.bell-labs.com/cm/cs/who/dmr/chist.html

W c

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*

Features of the C Lang.

- 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 Lang. (cont.)

- 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 Data Types

- **Basic types.**
- **Structured types.**
- **User defined types.**

Basic Types

- char
- int
- float
- double
- pointers

Optional specifiers*

- signed
- unsigned
- short
- long

***may not apply to all numeric types.**

char	long **int**
signed **char**	signed **long**
unsigned **char**	signed long **int**
short	unsigned **long**
short **int**	unsigned long **int**
signed **short**	long **long**
signed short **int**	long long **int**
unsigned **short**	signed long **long**
unsigned short **int**	signed long long **int**
int	unsigned long **long**
signed **int**	unsigned long long **int**
unsigned	**float**
unsigned **int**	**double**
long	long **double**

Wikipedia

Pointers

- A **variable** is a **memory location** into which **data** can be **stored**.
- C allows the programmer to access either the **variable location** or its **contents**.

C code excerpt:

```
int x;
int *px;

…

px = &x;
```

Integer variable.

Pointer to an integer variable.

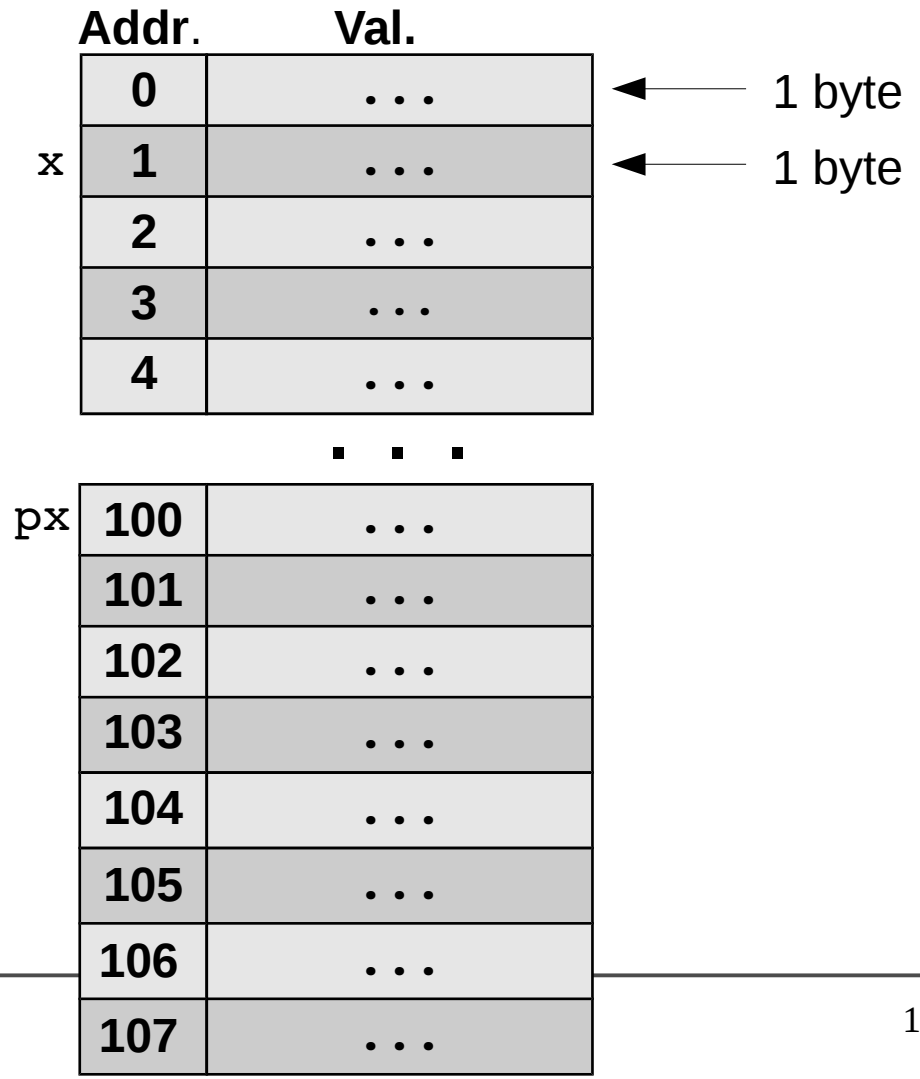
`px` receives the address of `x`.

Pointers

C code excerpt:

```
int x;
int *px;

…

x = 25;
px = &x;
```

RAM Memory



Pointers

C code excerpt:

```
int x;
int *px;

…

x = 25;
px = &x;
```

PC

00000000 00000000 00000000 00011001$_2$

RAM Memory

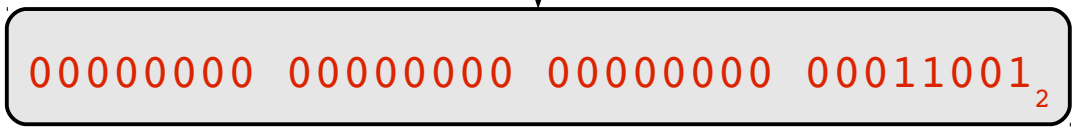
	Addr.	Val.
x	0	...
	1	...
	2	...
	3	...
	4	...
	...	
px	100	...
	101	...
	102	...
	103	...
	104	...
	105	...
	106	...
	107	...

Pointers

C code excerpt:

```
int x;
int *px;

…

x = 25;
px = &x;
```

PC



RAM Memory

	Addr.	Val.
	0	...
x	1	00011001
	2	00000000
	3	00000000
	4	00000000
	...	
px	100	...
	101	...
	102	...
	103	...
	104	...
	105	...
	106	...
	107	...

Little endian

Pointers

C code excerpt:

```
int x;
int *px;

…

x = 25;
px = &x;
```

PC

RAM Memory

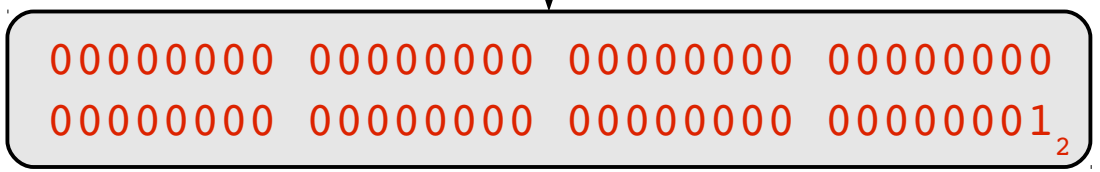
	Addr.	Val.
	0	...
x	1	00011001
	2	00000000
	3	00000000
	4	00000000
	...	
px	100	...
	101	...
	102	...
	103	...
	104	...
	105	...
	106	...
	107	...

Pointers

C code excerpt:

```
int x;
int *px;

…

x = 25;
px = &x;
```

PC



RAM Memory

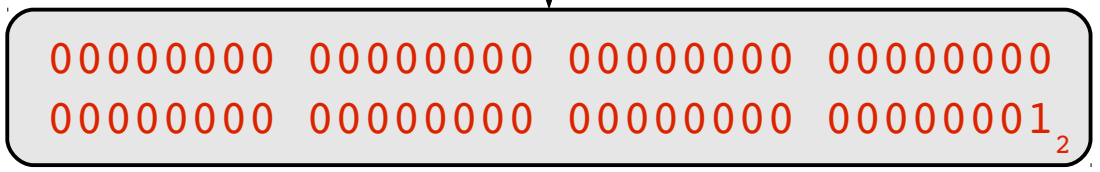
	Addr.	Val.
	0	...
x	1	00011001
	2	00000000
	3	00000000
	4	00000000
	...	
px	100	...
	101	...
	102	...
	103	...
	104	...
	105	...
	106	...
	107	...

Pointers

C code excerpt:

```
int x;
int *px;

…

x = 25;
px = &x;
```

PC



RAM Memory

	Addr.	Val.
	0	...
x	1	00011001
	2	00000000
	3	00000000
	4	00000000
	...	
px	100	00000001
	101	00000000
	102	00000000
	103	00000000
	104	00000000
	105	00000000
	106	00000000
	107	00000000

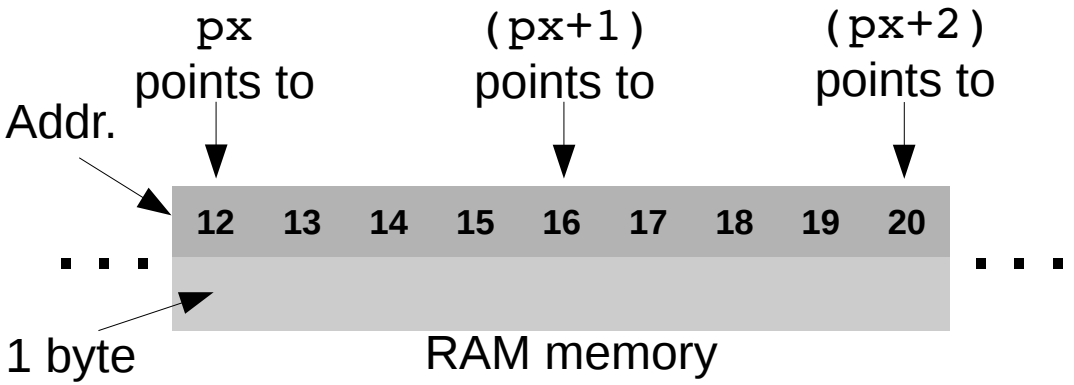
Why different pointer types?

- Mainly due two reasons:
  - To **reduce** the occurrence of **bugs**.
  - To allow for a **pointer arithmetic**!!!

Pointer Arithmetic

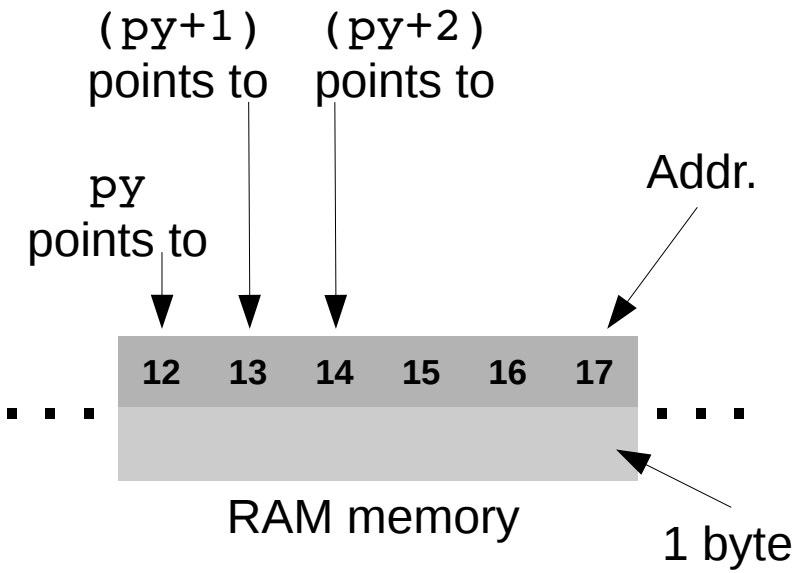
C code excerpts:

```
int *px;

*px = 2;
*(px+1)=-1; // or px[1]=-1
*(px+2)=5;  // or px[2]=5
```



C code excerpts:

```
char *py;

*py = 2;
*(py+1)=-1; // or py[1]=-1
*(py+2)=5;  // or py[2]=5
```



Hey! Have you asked **permission** to **write** to all those memory **positions???**

Memory Allocation Policies

- Memory can be allocated:
  - **Automatically**.
  - **Statically**.
  - **Dynamically**.

Automatic Variables

- Memory is **automatically allocated** and **deallocated** when the program execution '**enters**' or '**leaves**' the lexical variable **scope**.
- **Example:**

```
 2
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5
 6 void loop(void)
 7 {
 8     int a;
 9
10     for (a=0; a<10; a++)
11         printf("%i\n",a);
12 }
13
14 int main(void)
15 {
16     loop();
17
18     return 0;
19 }
20
```

Static memory Allocation

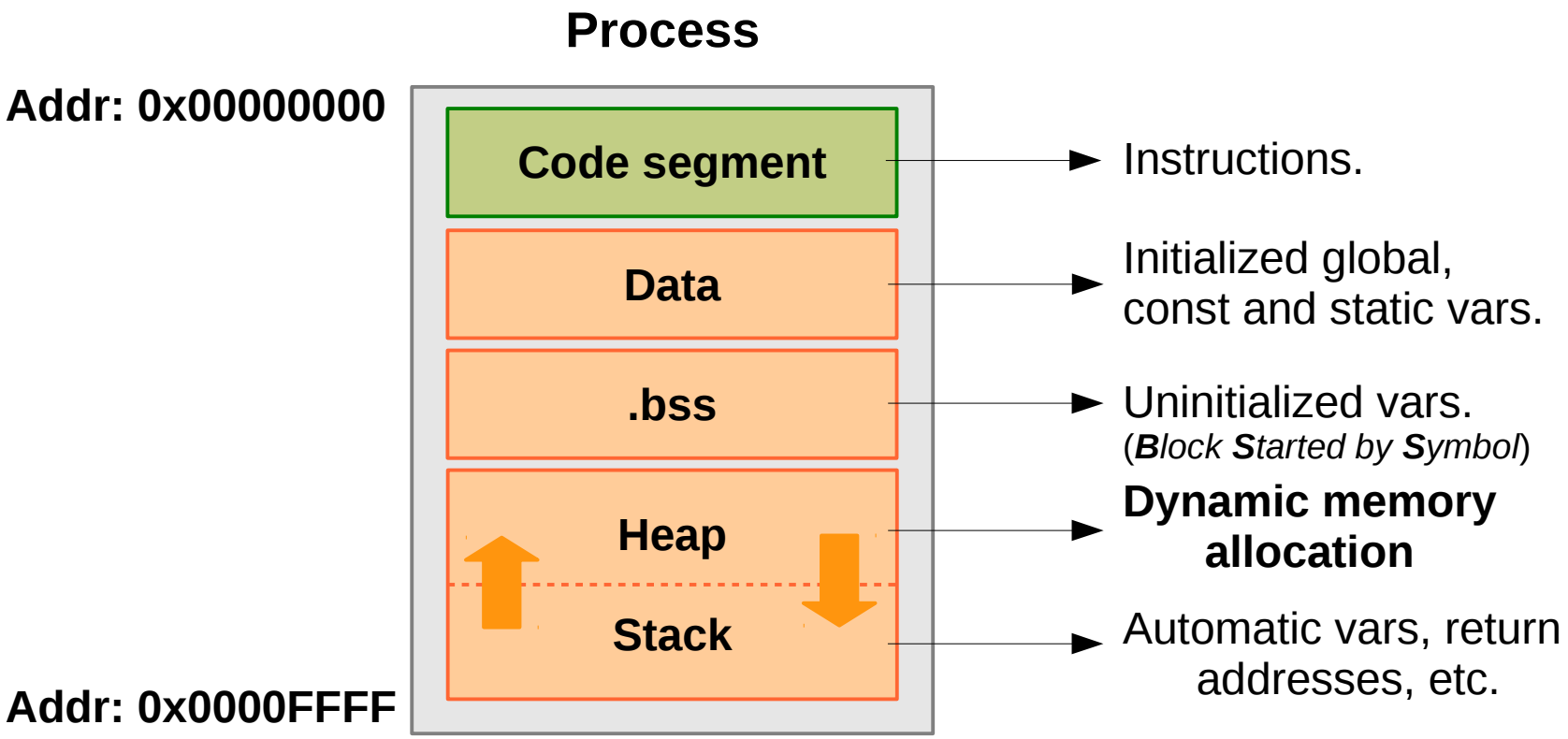
- Memory is **allocated** at **compile time**.
- Memory remains allocated during the entire **program life time**.
- **May** have **limited scope** (e.g. static local).

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

void test(void)
{
    int a = 0;
    static int b = 0;

    printf("Initial a value: %i\n", a);
    printf("Initial b value: %i\n", b);

    a++;
    b++;

    printf("Final a value: %i\n", a);
    printf("Final b value: %i\n", b);
}

int main(void)
{
    printf("\n");

    test();

    printf("----------------------\n");

    test();

    return 0;
}
```

C code example...
(static_var.c)

Dynamic Memory Allocation

- Approximated **process memory layout** (in the **virtual memory space**):



Allocating Space in the Heap

- C functions allows for the allocation/deallocation of heap memory:
  - ***malloc***()
    - Allocate X bytes and returns a pointer to the first byte of allocated memory.
  - ***calloc***()
    - Allocate space for X array elements, initializes to zero, and return a pointer to the allocated memory.
  - ***free***()
    - Deallocate previously allocated memory.
  - ***realloc***()
    - Change the size of previously allocated memory.

Building an Array with *malloc*()

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

int main(void)
{
    int *pvi = NULL;
    int i;

    printf("\n");

    printf("Initial pvi: %p\n", pvi);

    pvi = (int*) malloc(10 * sizeof(int));

    printf("After malloc pvi: %p\n", pvi);


    for (i=0; i<10; i++)
        printf("Value at position %i: %i\n", i, pvi[i]);


    for (i=0; i<10; i++)
        *(pvi+i) = -i * 10;

    printf("After update pvi: %p\n", pvi);

    printf("-----------------------------\n");

    for (i=0; i<10; i++)
        printf("Value at position %i: %i\n", i, pvi[i]);

    printf("Before free() pvi: %p\n", pvi);

    free(pvi);

    printf("After free() pvi: %p\n", pvi);

    return 0;
}
```

C code example...
(malloc.c)