The C Programming Language

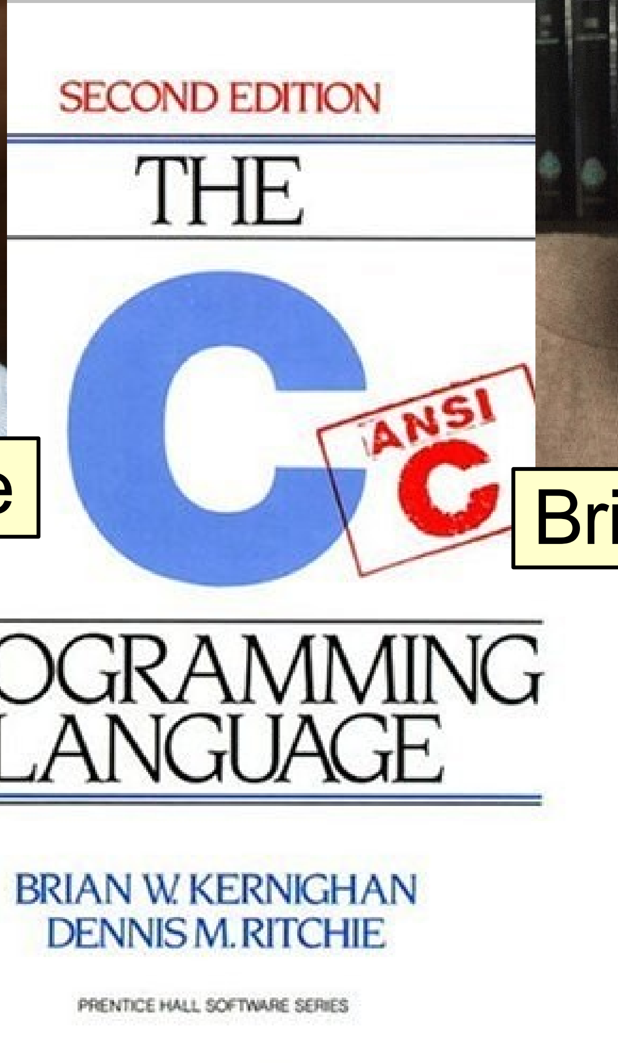
CS 241: Systems Programming
Discussion, Week 1

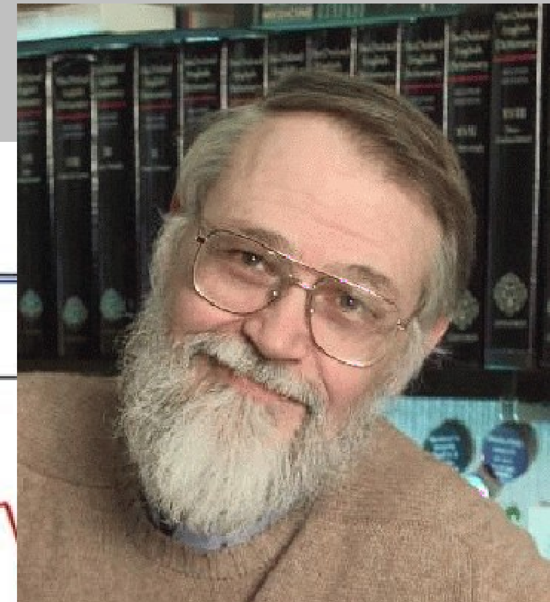
slides prepared by Sameer Sundresh

The C Programming Language

Ken Thompson

Dennis Ritchie





Brian Kernighan

Bell Laboratories

Overview

Syntax

Memory model

Discussion Problems

The C Programming Language

C and Unix grew up together

Native Unix systems programming language

Unix, Linux and Windows are ***implemented*** in C

Essentially a ***portable assembly language***

A canonical C program

```
/* hello.c */
#include <stdio.h>


int main(int argc, char **argv)
{
  fprintf(stdout, “Hello, CS %d \n”, 241);
  return 0;
}
```

A canonical C program

```
/* hello.c */
#include <stdio.h>

int main(int argc, char **argv)
{
  fprintf(stdout, "Hello, CS %d \n", 241);
  return 0;
}
```

comment

preprocessor directive

main entrypoint

argument types

pointer type

return type

standard I/O library function

format string

Things to notice

Very similar syntax to C++ and Java

C++ was originally a front-end to C (in the '80s)

But there are differences:

- No objects, classes, inheritance or namespaces

  just functions and data

  *use **arrays** and **structs** to make data structures*

- No templates
- No operator overloading (<< means shift)
- No exceptions: ***always check return values!***

When you need a reference: manual pages

Unix manual pages (man pages)

man [section-number] <command-or-function>

man gcc

man 3 fprintf

Relevant manual sections

1 General commands (that you run from the shell)

2 System calls (C functions provided by OS kernel)

3 C library functions

Be aware: sometimes a command and a system call or library function have the same name.

Problem 1: using fprintf

Let

```
float x = 1234.1234;
int y = 1234;
```

```
man 3 fprintf
```

```
/* hello.c */
#include <stdio.h>

int main(int argc, char **argv)
{
  fprintf(stdout,
    "Hello, CS %d \n", 241);
  return 0;
}
```

Output:

```
Problem #1
```

(value of x as a float)

(y in decimal) (y in hexadecimal)

(without the parentheses)

Overview

Syntax

Memory model

Discussion Problems

C language source code files

It's where you write your programs.

Layout of a .c file

includes

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

type definitions

```
typedef char *string;
```

function declarations

```
int main(int argc, char **argv);
```

function definitions

```
int main(int argc, char **argv) { ... }
```

C language header files

It's how you share code between source files.

Layout of a .h file

includes

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

type definitions

```
typedef unsigned int uint32_t;
```

function declarations

```
void exit(int);
```

...but no function definitions

Arrays and pointers

Array types

type arr[*dim1*][*dim2*]...

Pointer types *(more on pointers in a moment)*

type *   *type* **   ...

A pointer specifies the ***memory address*** of a value or the ***first address*** of an *array* of values.

Examples

```
char *
```

A C string is just a pointer to a sequence of one or more characters in memory.

```
int a[8], b[2][4];
char *argv[], **also_argv;
```

Overview

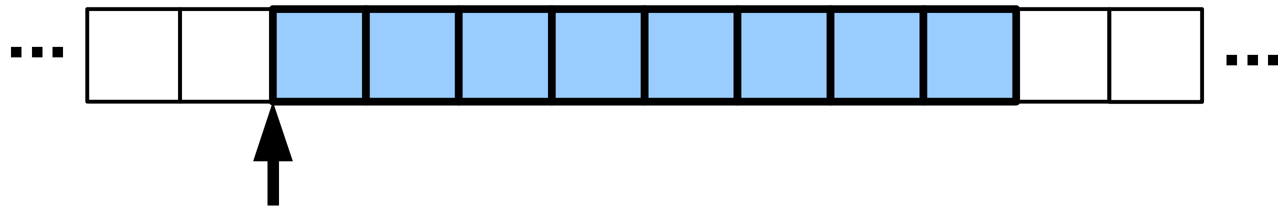
Syntax

Memory model

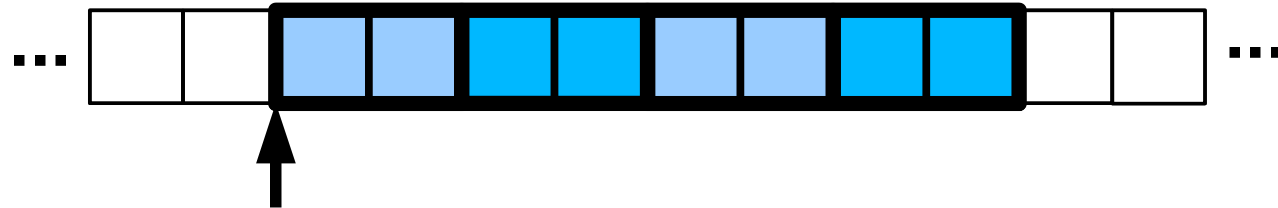
Discussion Problems

Arrays

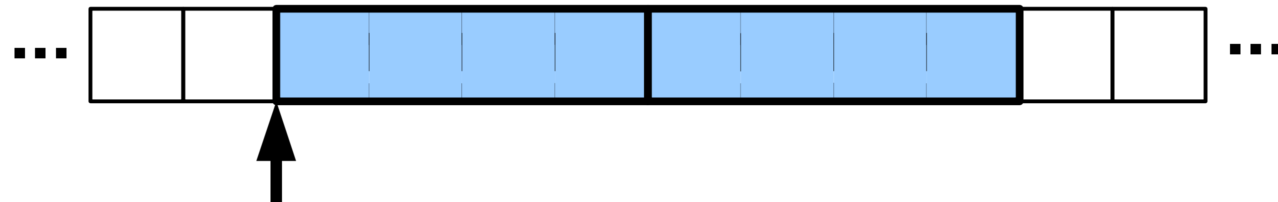
```
char a[8];  /* array of bytes */
```




```
char b[4][2];  /* 2-dimensional array */
```




```
int c[2];  /* array of 32-bit words */
```



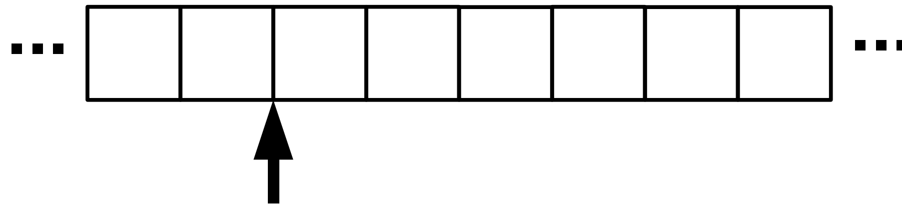
Memory

Memory is just a big array of bytes

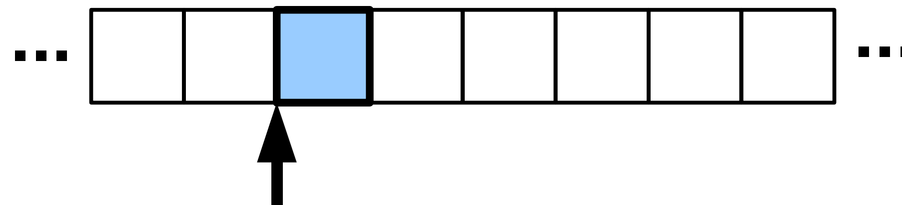
Pointers are indicies into memory

The ***type*** of a pointer determines whether the memory it indexes is viewed as a `char`, an `int`, etc.
`void` indicates the *no-value* type.

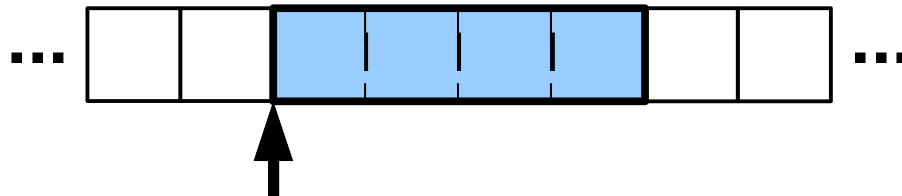
```
void *p = ...;
```



```
(char *) p
```



```
(int *) p
```



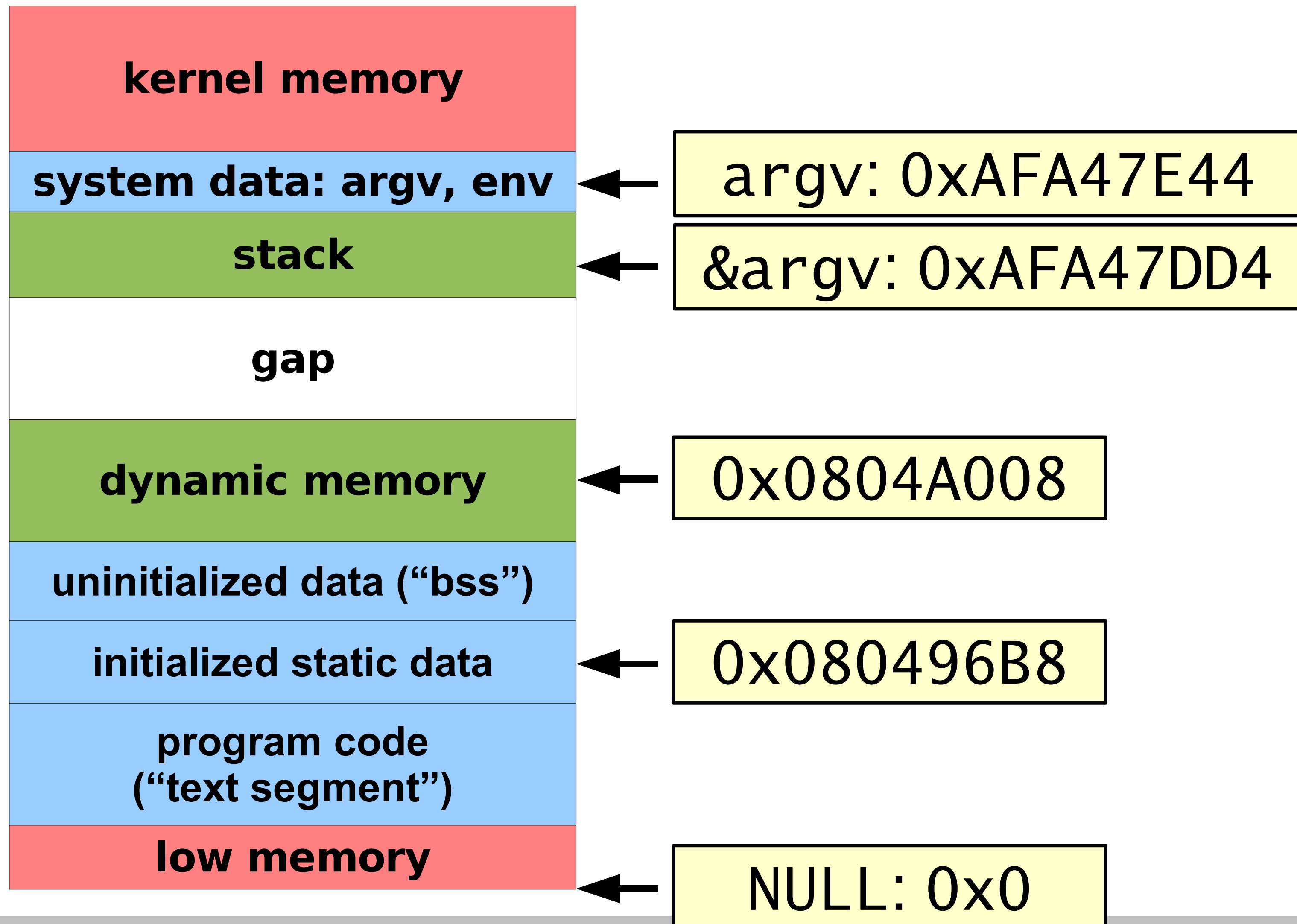
Referencing and dereferencing

The & operator creates a pointer to a variable *(takes the address of the memory location holding the variable)*, while the * operator reads the data which a pointer references:

```
int x;
int *xptr = &x;
/* xptr = 0xAF981DF8 */

int y = *xptr;
/* y = x */
```

Process memory layout



Kinds of memory allocation

Static allocation

Variables declared at top level of program (outside of any function).

Stack allocation

Variables declared within a function

Dynamic allocation

`malloc`(*size*) → *pointer to allocated memory*

`free`(*ptr*) → **ptr is no longer allocated*

Problem 2: What's wrong with this program?

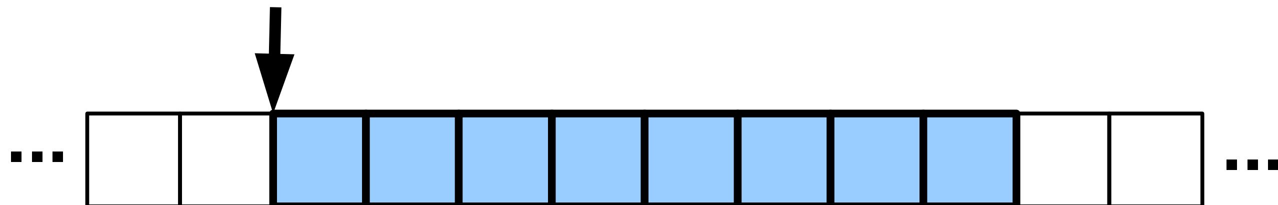
```
#include <stdlib.h>

int main(int argc, char **argv)
{
    int **buf = (int **) malloc(10*sizeof(int));
    for (int i = 0; i < 10; i++) {
        buf[i] = (int *) malloc(i*sizeof(int));
    }
    free(buf);
    return 0;
}
```

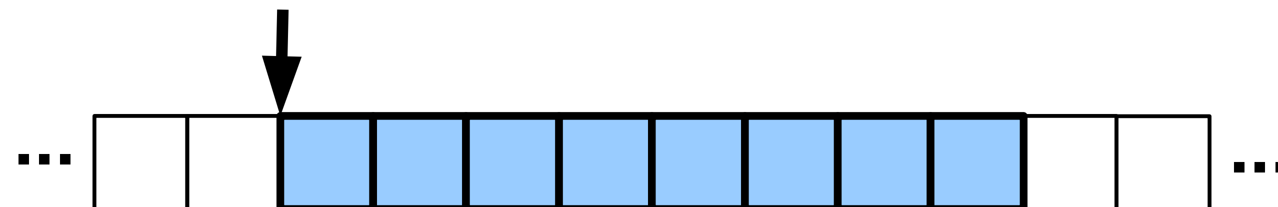
Create a bugfixed version.

Pointer arithmetic

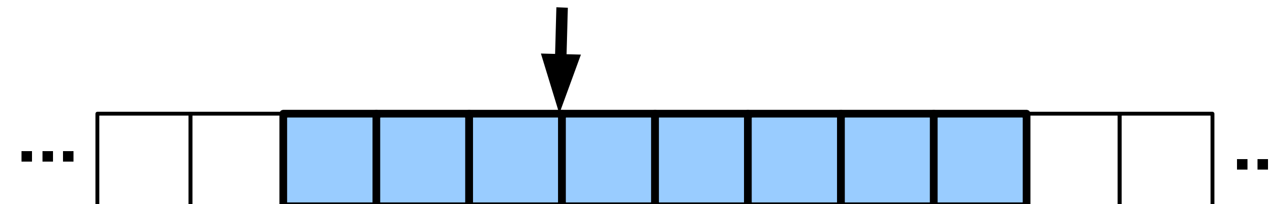
```
char a[8];      /* array of bytes */
```




```
char *p = a;    /* p, a: 0xAF99EFDC */
```

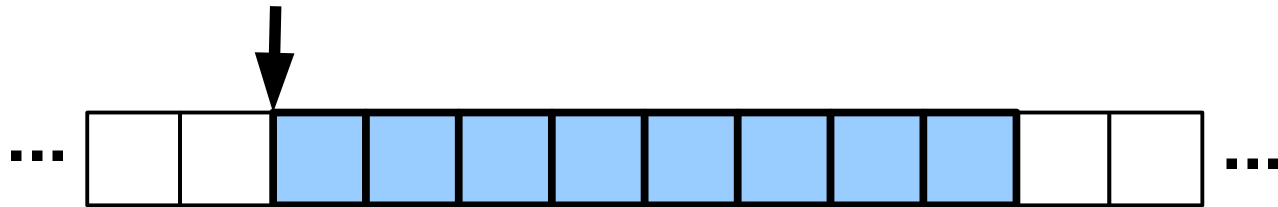



```
char *q = a+3; /* q: 0xAF99EFDF */
```

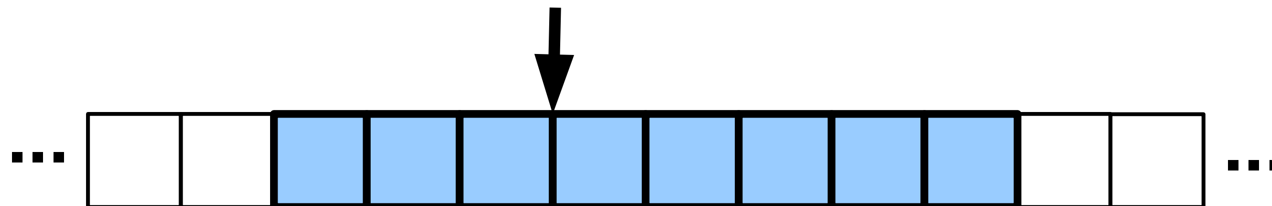


Pointer arithmetic (2)

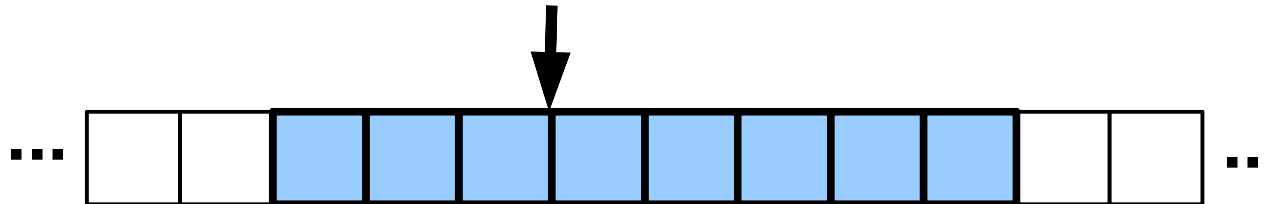
```
char a[8];         /* array of bytes */
```




```
char *q = a+3;     /* q: 0xAF99EFDF */
```

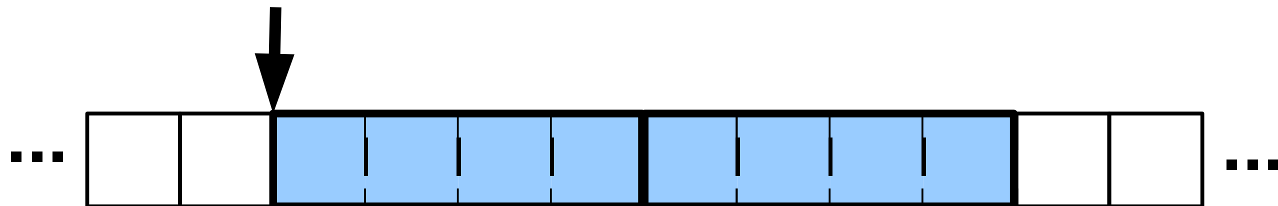



```
char *r = &a[3]; /* r: 0xAF99EFDF */
```

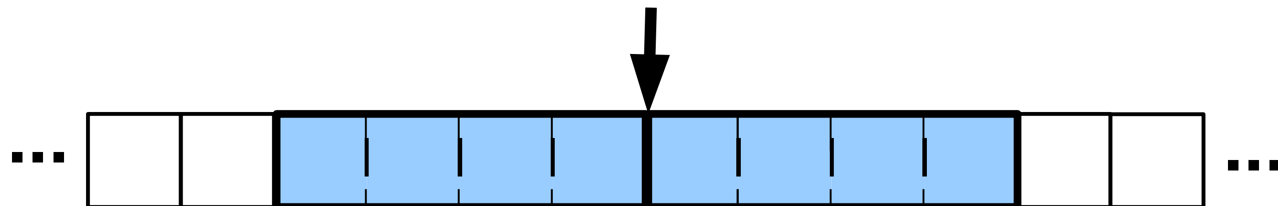


Pointer arithmetic (3)

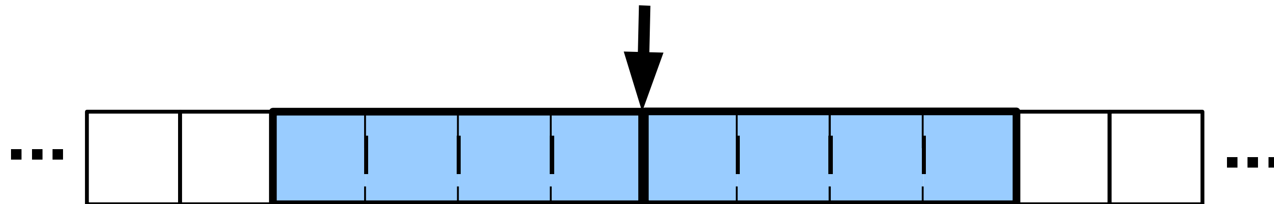
```
int b[2];  /* array of 4-byte words */
```




```
int *q = b+1;      /* q: 0xAF99EFE0 */
```




```
char *r = &b[1]; /* r: 0xAF99EFE0 */
```



Overview

Syntax

Memory model

Discussion Problems

Problem 3: What's wrong with this program?

```
#include <string.h>

char *twiceA(char str[10])
{
    char buf[20];
    int n = strlen(str);
    strncpy(buf, str, n);
    strncpy(buf+n, str, n);
    return buf;
}
char *twiceB(char *str)
{
    int n = strlen(str);
    return strncpy(str+n, str, n);
}
```

Hint

a string is a pointer to a sequence of characters, terminated by a null: `(char) 0` *or* `'\0'`

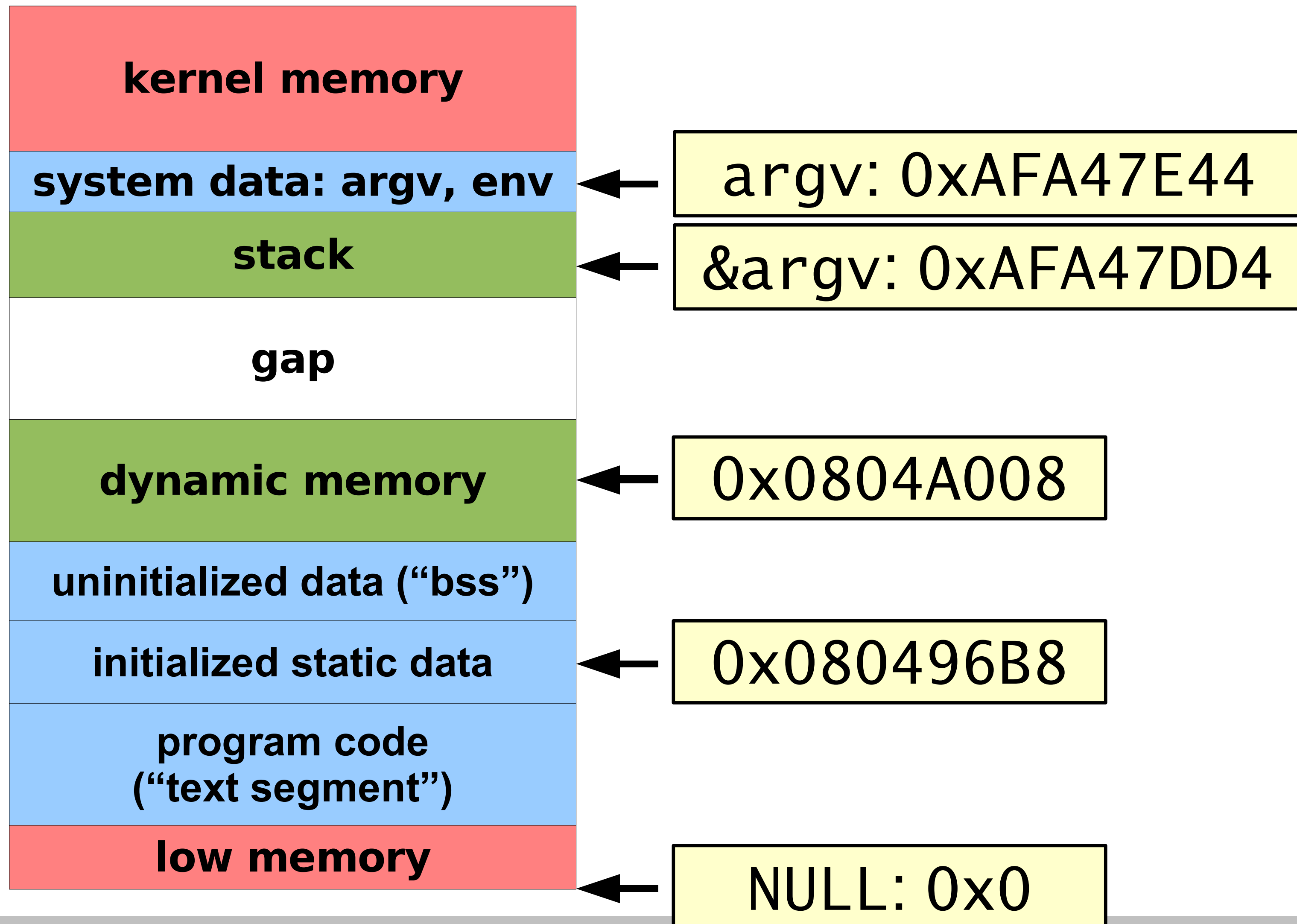
Problem 4: Check memory layout

Print out the *addresses* of the following:

`argv`	*pointer to argument vector on stack*
`argv[0]`	*pointer to first argument*
`*argv[0]`	*first character of first argument*
`main`	*program entrypoint function*
-	*a global variable (static data)*
-	*a local variable within a function*
-	*a dynamically-allocated buffer*

Hint: use format string “0x%08X” for a pointer.

Process memory layout



Problem 5: Layout of argv

Print out the *contents* of:

```
argv
```

`argv[0]`, ..., `argv[argc-1]`

Hint: the *contents* of `argv` is an array of `argc`-many pointers of type **`char *`**

Draw a diagram of what it looks like.

Layout of argv