Memory Allocation and Scope

Lecture 4

Christian A. Pagot

Universidade Federal da Paraíba
Centro de Informática

Physical Memory (RAM)

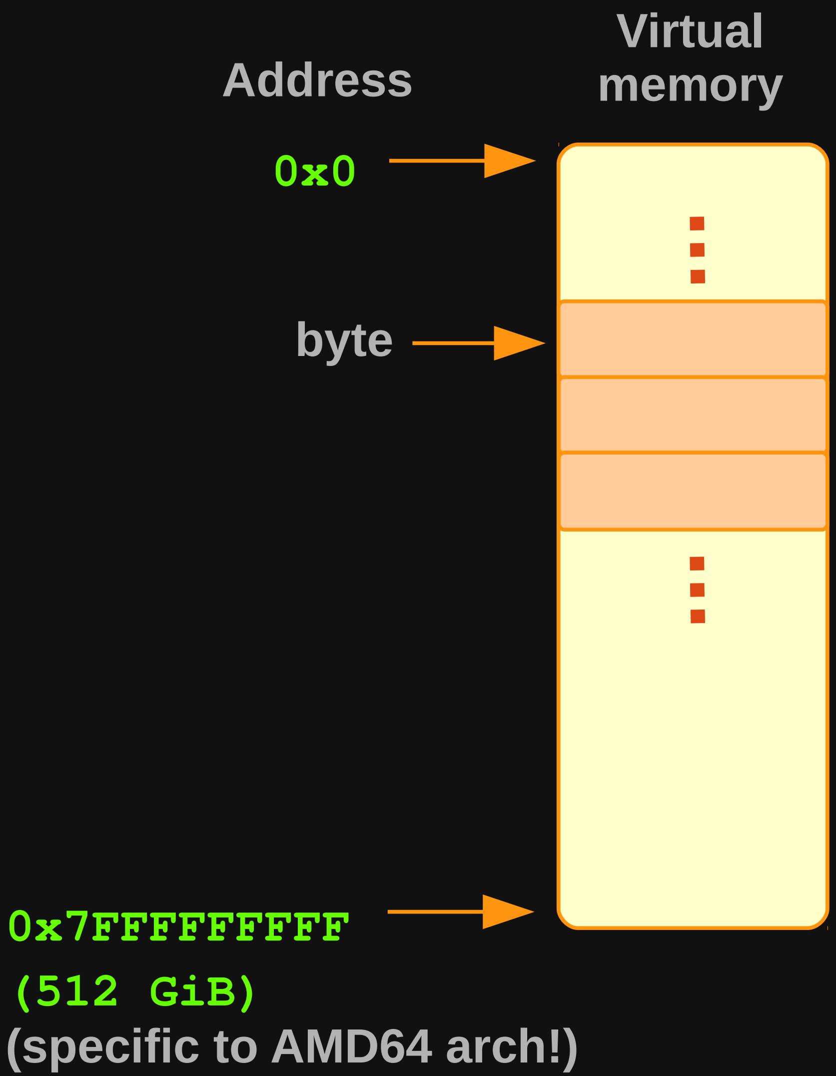
- May not be **sufficient** to fulfill our needs.
- Certain **code** portions may be **rarely used**.
- Physical memory should be **abstracted**.
- Usually, processes **should not interfere** with each other.

How we could **handle** these **problems/limitations**?

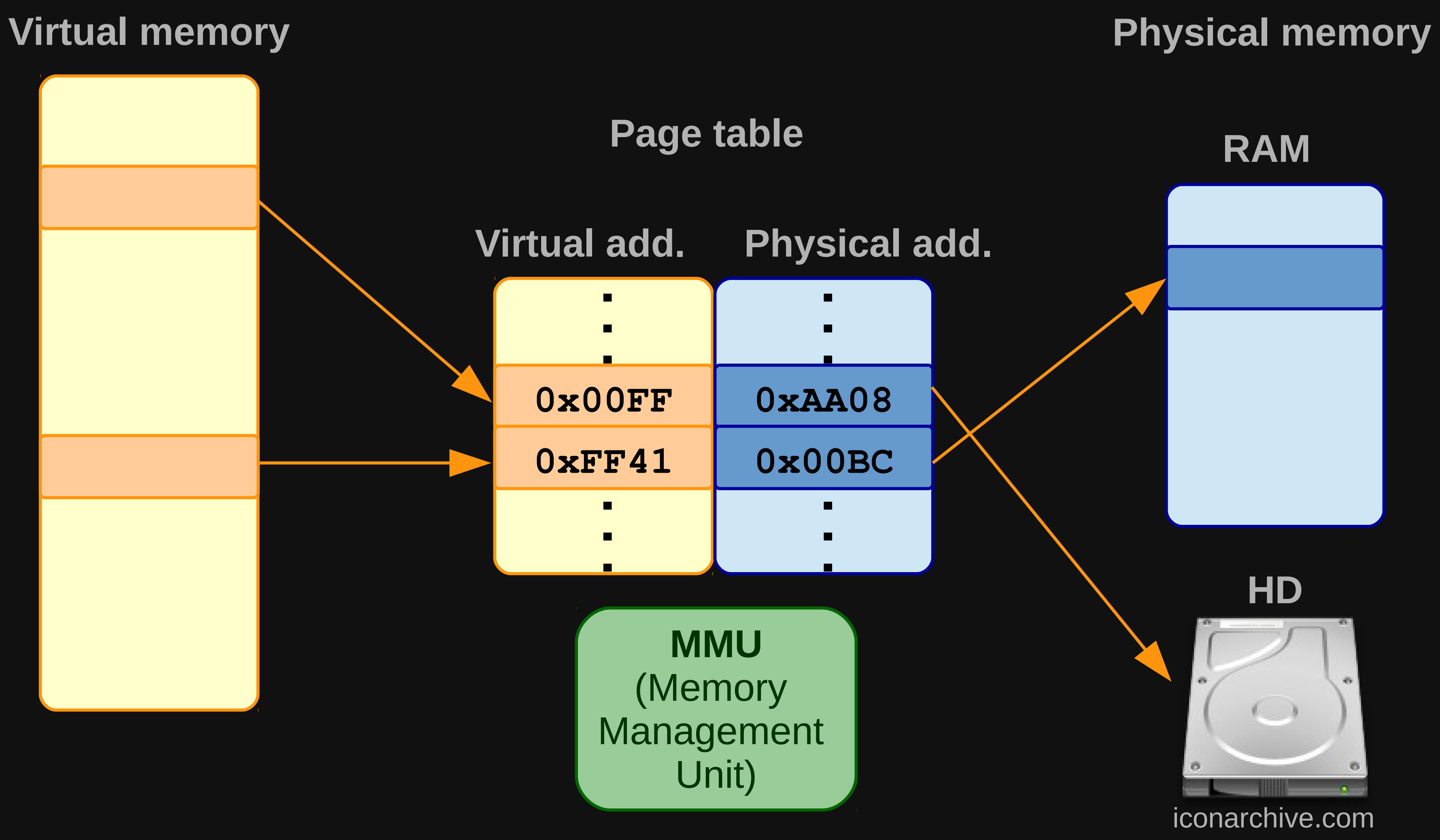
Virtual Memory!

Virtual Memory

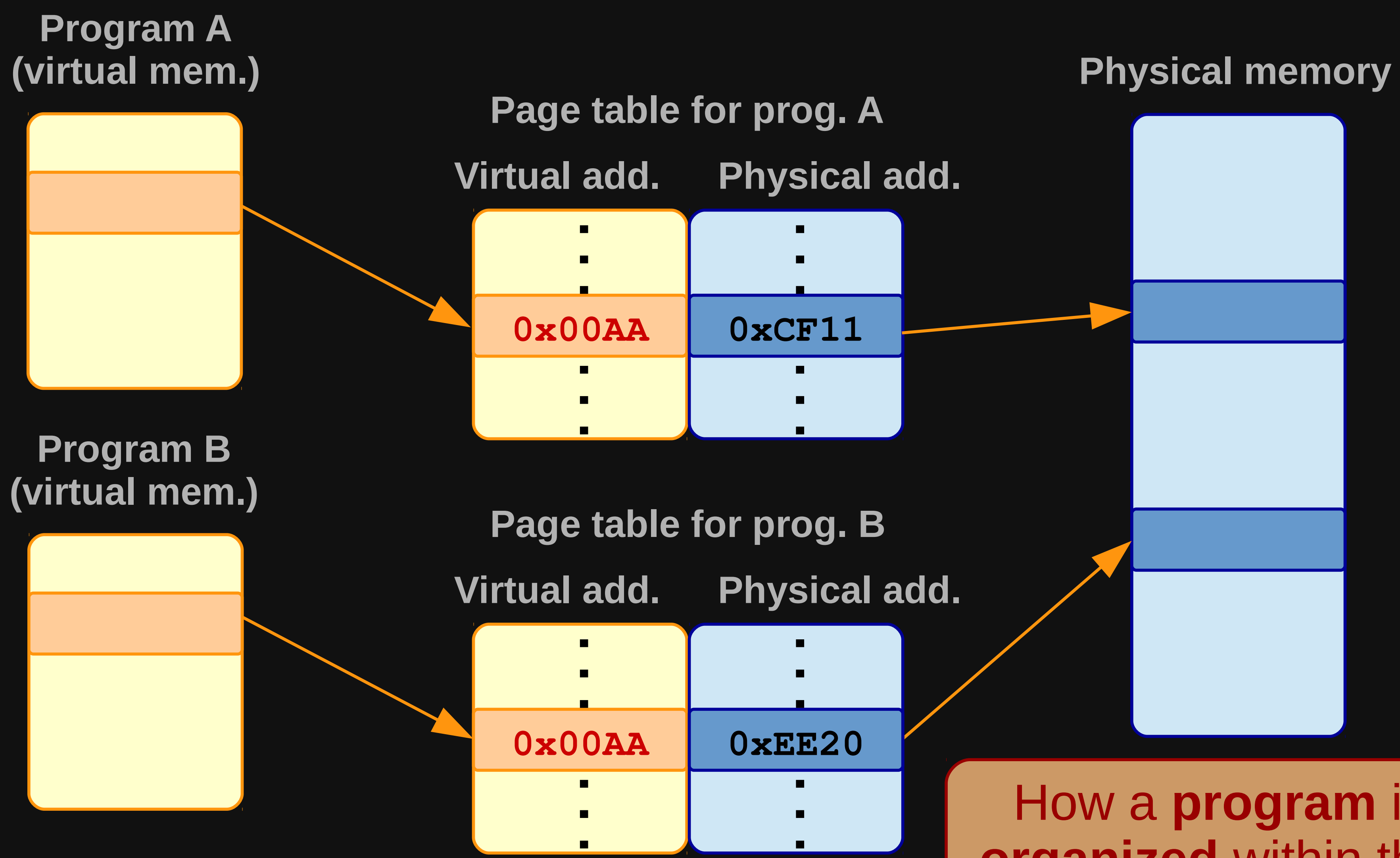
- Main storage appears as a **continuous** address space.
- Maps **virtual** addresses to **physical** addresses.
- The OS **manages** virtual address spaces.
- Address translation is accomplished by the CPU, through dedicated hardware: **Memory Management Unit** (MMU).
- Even **files** (located on the HD) can be **mapped** into the virtual memory!



Virtual Memory



Virtual Memory



How a **program** is **organized** within the **virtual memory**?

Physical / Virtual Memory Example

Checking current physical / virtual memory usage

```
~$ man top
```

```
~$ top
top - 16:14:06 up  2:19,  2 users,  load average: 0,57, 0,52, 0,46
Tasks: 232 total,   1 running, 231 sleeping,   0 stopped,   0 zombie
%Cpu(s):  7,8 us,  2,0 sy,  0,0 ni, 90,0 id,  0,0 wa,  0,0 hi,  0,2 si,  0,0 st
KiB Mem:   3949960 total,  2980252 used,   969708 free,    96540 buffers
KiB Swap: 22142972 total,        0 used, 22142972 free.  1168940 cached Mem
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1323 root      20   0  250052  84100  46968 S   9,6  2,1   8:10.86 Xorg
 2371 christi+  20   0 1267512 246364 133924 S   9,6  6,2   7:18.72 chrome
 3272 christi+  20   0 1044068 232076  77696 S   7,6  5,9   4:11.55 chrome
 2082 christi+  20   0 1390992 191244  84784 S   6,0  4,8   3:18.66 compiz
```

`VIRT` : virtual memory size (code, data, shared libraries, etc.).
`RES` : resident memory size (non-swapped physical memory).
`SWAP` : swapped size (non-resident portion of the task's address space).
`SHR` : memory potentially shared with other processes.

`top`**:** press '**f**' to **select columns**!

DIY!

Storage Class and Scope

- Programs are composed of:
  - Executable code.
  - Data.
- Data
  - Pertains to a **storage class**, which defines its **lifespan**.
  - Has a **scope**, which defines its **visibility** within the program.
  - “*Storage class and scope are assumed from the location of a datum's declaration, and determine its placement within virtual memory.*”

Storage Class and Scope

- Data declared **outside** any **function** has
  - Global scope.
  - Static duration (it exists for the program lifespan).
- Example

example_31.c

```c
int x = 7;

void f() {
    x = 19;
}

int main() {

    x = 13;
    f();

    return 0;
}
```

- `x` has **global scope.**
- `x` has **static duration.**

Storage Class and Scope

- Data declared **inside a function**
  - Local scope.
  - Automatic duration (it exists for the duration of the function call).
- Example

example_32.c

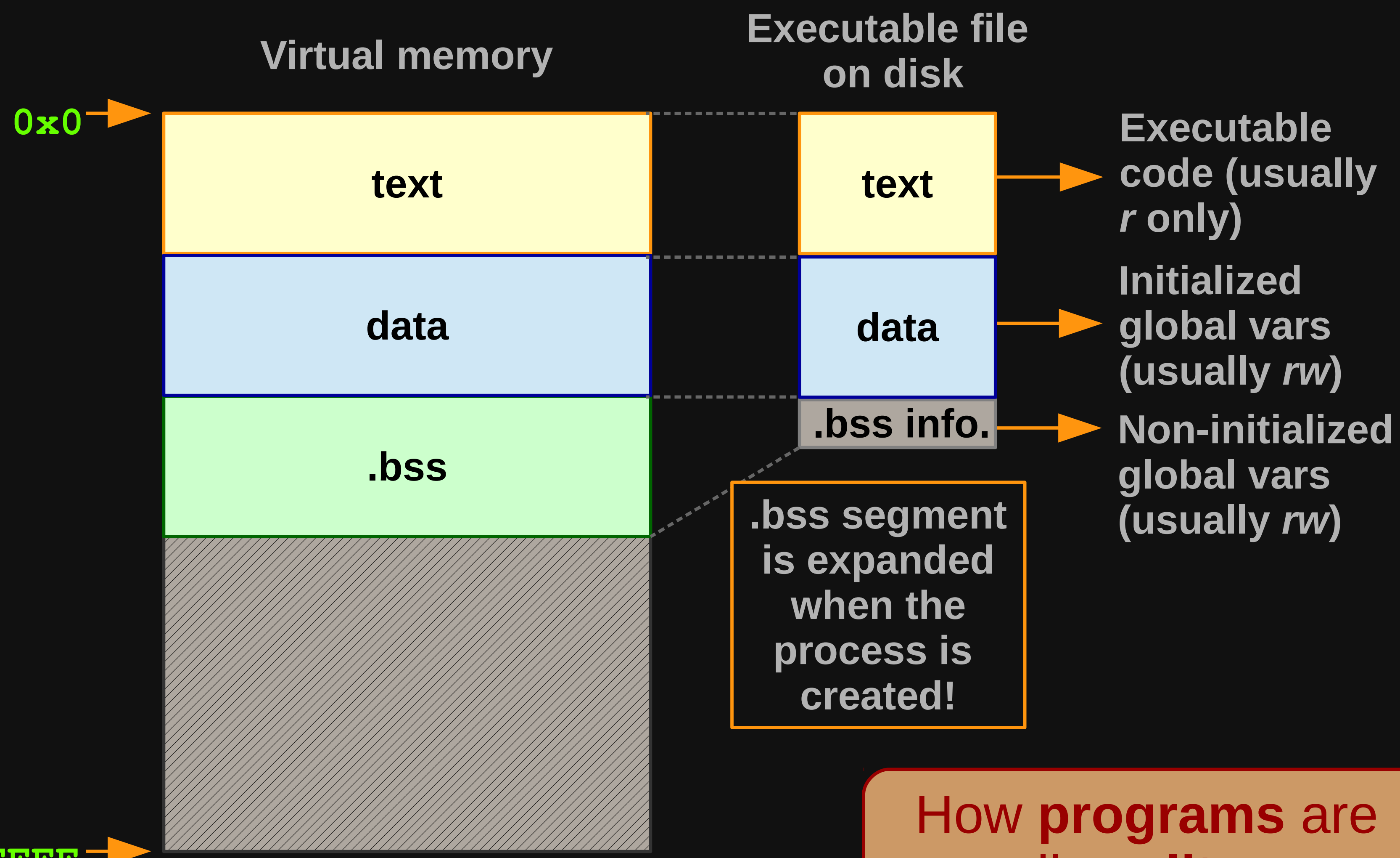
```
void f() {
    int x = 19;
    x++;
}

int main() {

    f();
    f();

    return 0;
}
```

- **x** has **local scope.**
- **x** has **automatic duration.**

Programs vs. Processes

- Program
  - Source file.
  - Compiled source file on disk.
- Process
  - "*Instance of a computer program that is being executed*" (Wikipedia).
  - Consists of:
    - PID (Process IDentification number).
    - Context (register values).
    - Page table.
    - List of open files, etc.

Process Memory Layout



(specific to AMD64 arch!)

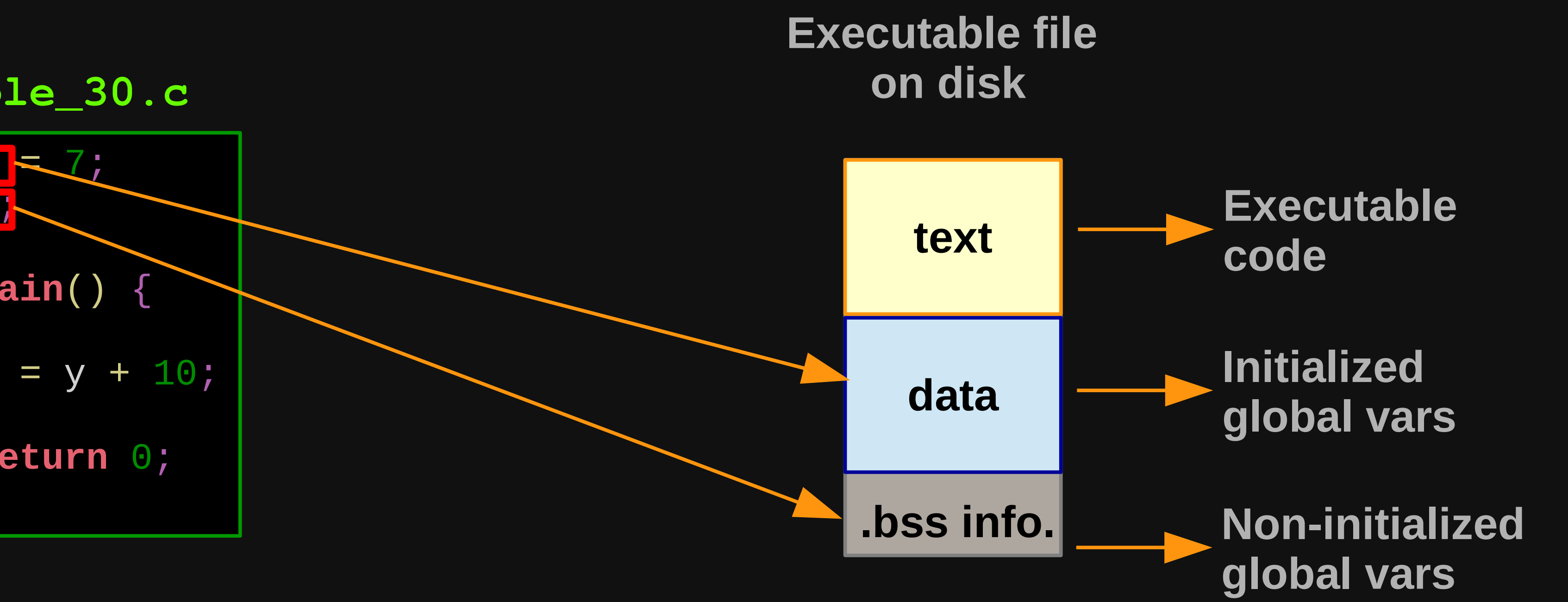
How **programs** are actually **split** across these **segments**?

Program Segmentation Example

- **Executable and Linking Format** (ELF) is the executable file format used by **Linux**.
- **Data** and **instructions** are **aggregated** by the compiler in distinct **segments**.

example_30.c

```c
int x = 7;
int y;

int main() {

    x = y + 10;

    return 0;
}
```



Process Memory Layout Example

Example

DIY!

example_34.c

```
char x[104857600];

int main (void)
{
    return 0;
}
```

```
~$ gcc example_34.c -o example_34
~$ ls -l
-rwxrwxr-x 1 christian christian   8532 Jan 29 20:30 example_34
-rw-rw-r-- 1 christian christian     55 Jan 29 20:29 example_34.c
```

```
~$ man size
```

```
~$ size example_34
   text     data       bss      dec      hex filename
   1115      552 104857632   104859299   64006a3 example_34
```

More on Storage Class and Scope!

- Data with **local scope** can present **static duration** if declared with the `static` **qualifier**.
  - It is important when we want to retain the value of a variable between function calls!
- Example

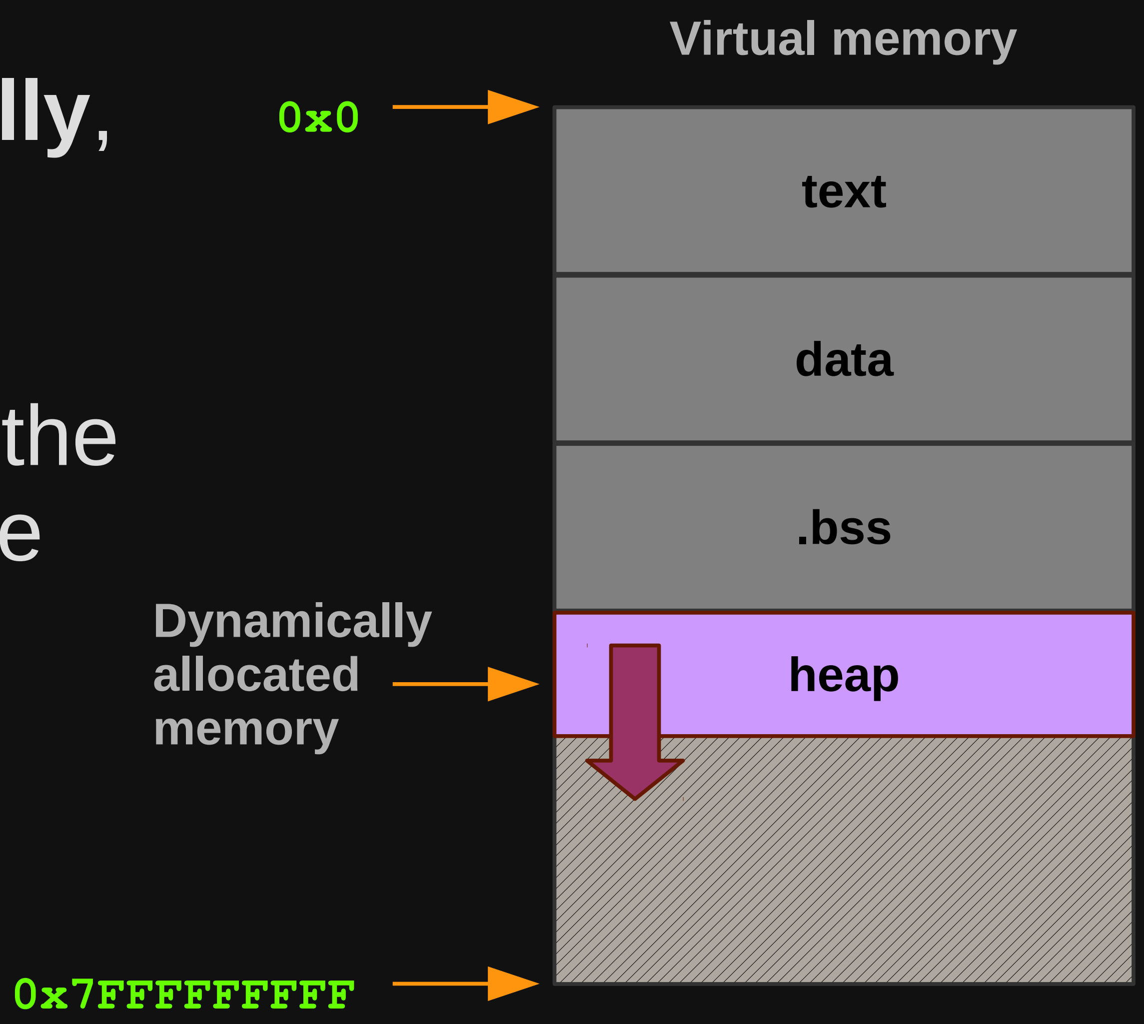
example_33.c

```
void f() {
    static int x = 19;
    x++;
}

int main() {

    f();
    f();

    return 0;
}
```

- `x` has **local scope.**
- `x` has **static duration.**

Dynamic Memory Allocation

- Memory can be allocated **dynamically**, during **runtime**.
- Dynamic memory is allocated right **after** the **data segment**, in the **heap segment**.

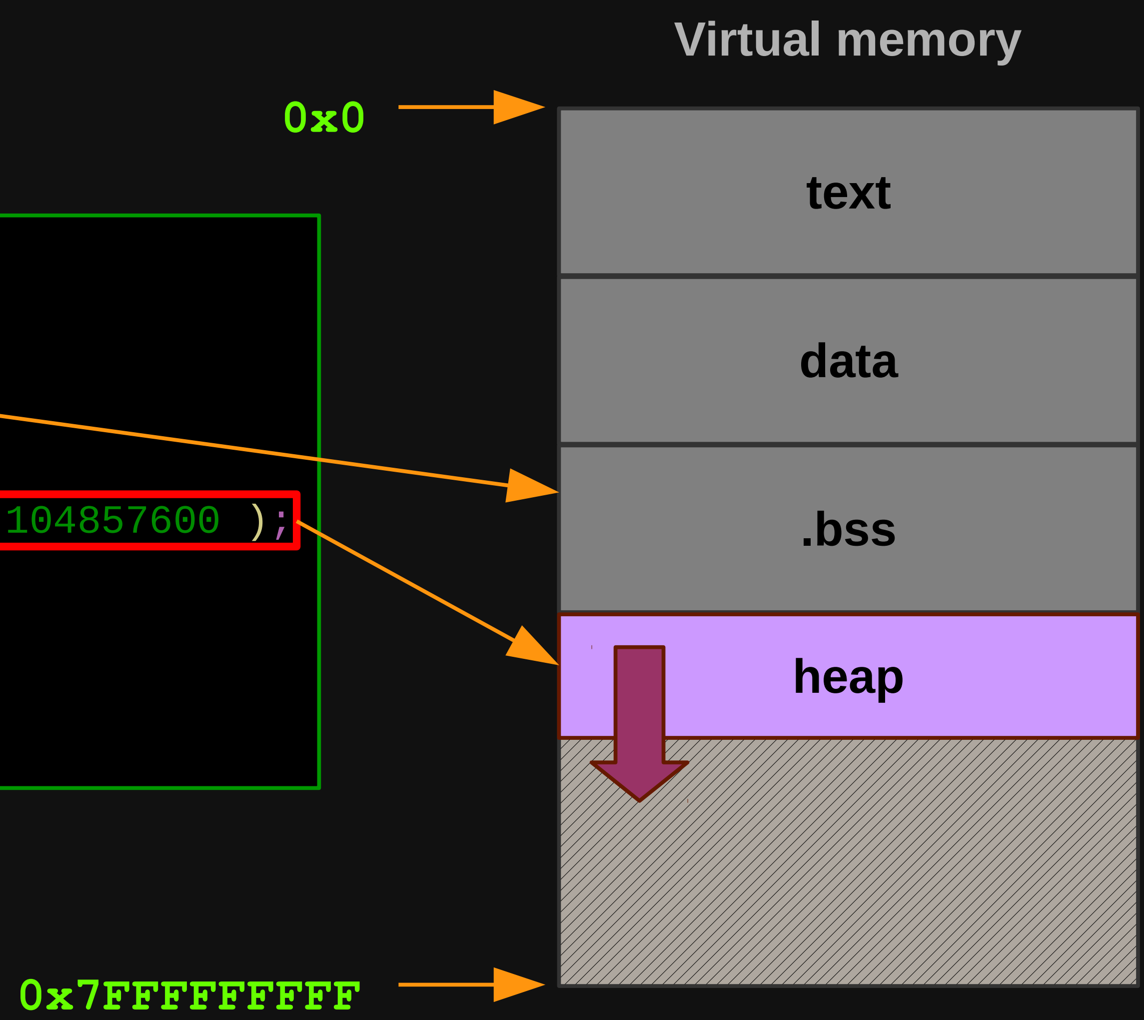


Dynamic Memory Allocation

Example

example_35.c

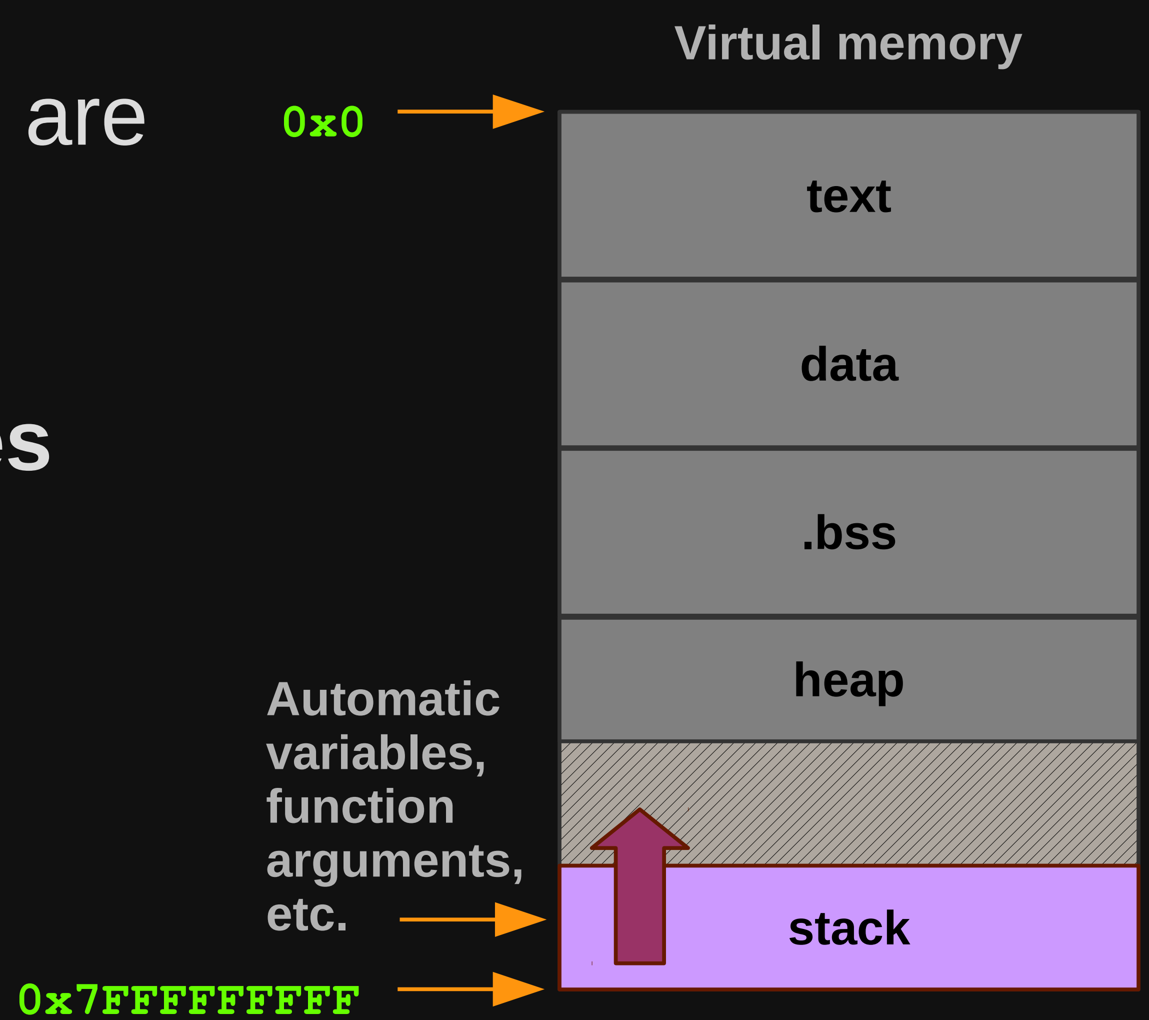
```
#include<stdlib.h>

char * pc;

int main (void)
{
    pc = malloc( sizeof( char ) * 104857600 );

    free( pc );

    return 0;
}
```



Automatic Variables

- **Local** variables and function **arguments** are allocated **automatically**.
- **Automatic variables** are pushed onto the **stack**.

More on **parameter passing** later!

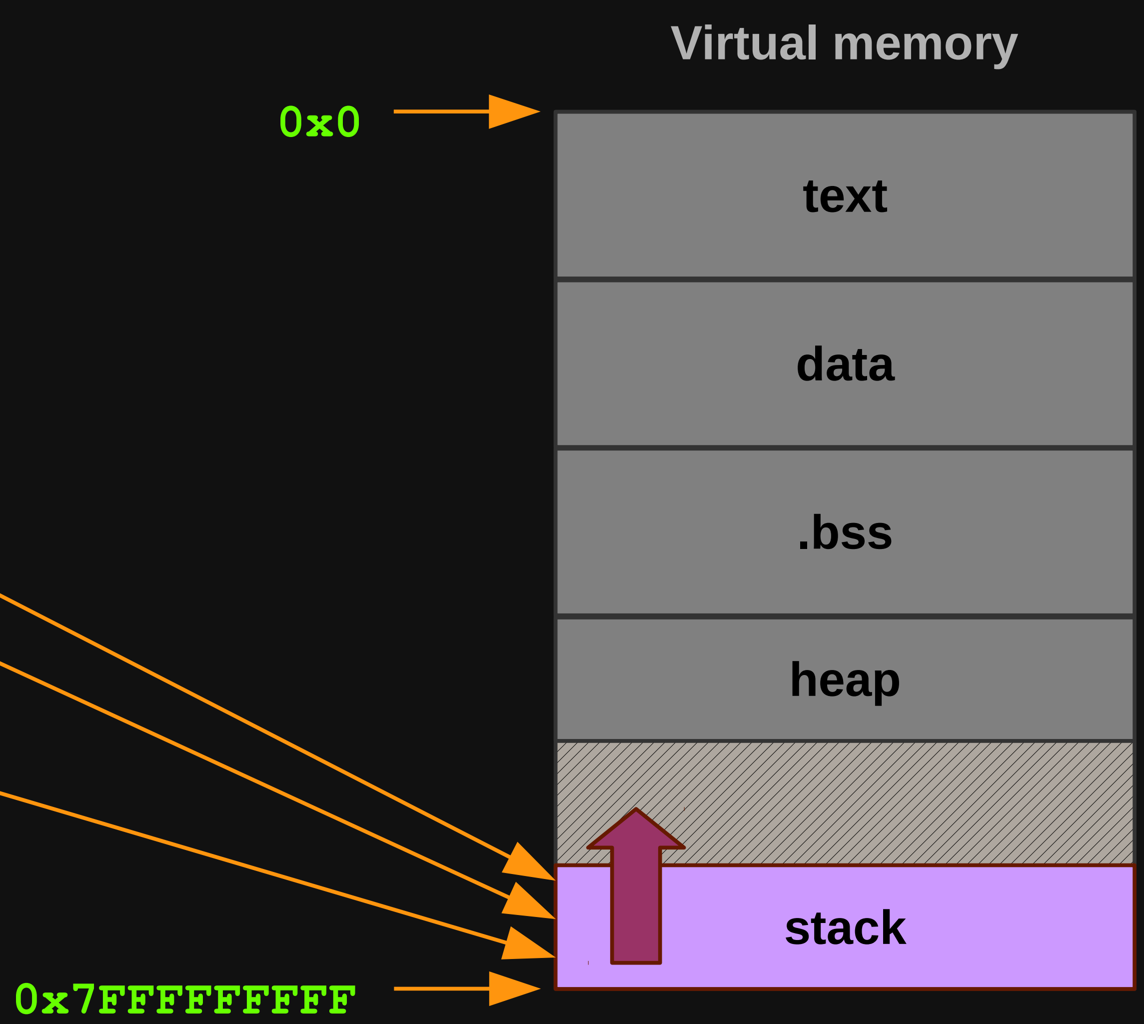


Automatic Variables

Example

example_37.c

```c
#include <stdio.h>

void f( int i ) {
    int y = i + 20;
    printf( "value: %i\n", y );
}

int main() {
    int x = 10;
    f( x );

    return 0;
}
```



Stack Facts

- GNU compilers allow memory to be allocated **dynamically** on the current **stack frame** with **`alloca()`**.

  There is no need to explicitly free **`alloca()`**'ted memory. It is automatically released at the end of the current function call.

- Stacks usually have a **maximum** prescribed **size**.

  Thus, take care when using **`alloca()`**!

- Stacks allow for **recursion**!

More on “*the **terrible story** of **`alloca()`** and the **`inline`** function*” later!

C99 Variable Length Array (VLA)

- It's an **array type** whose **size** is determined only at **runtime**.

  Exercise:

  - Try using GDB to find how the VLA is **allocated on your system**.
  - Could you describe at least one **difference** between the VLA and **`alloca()`**?

Parameter Passing

- C supports **by-value** parameter passing!

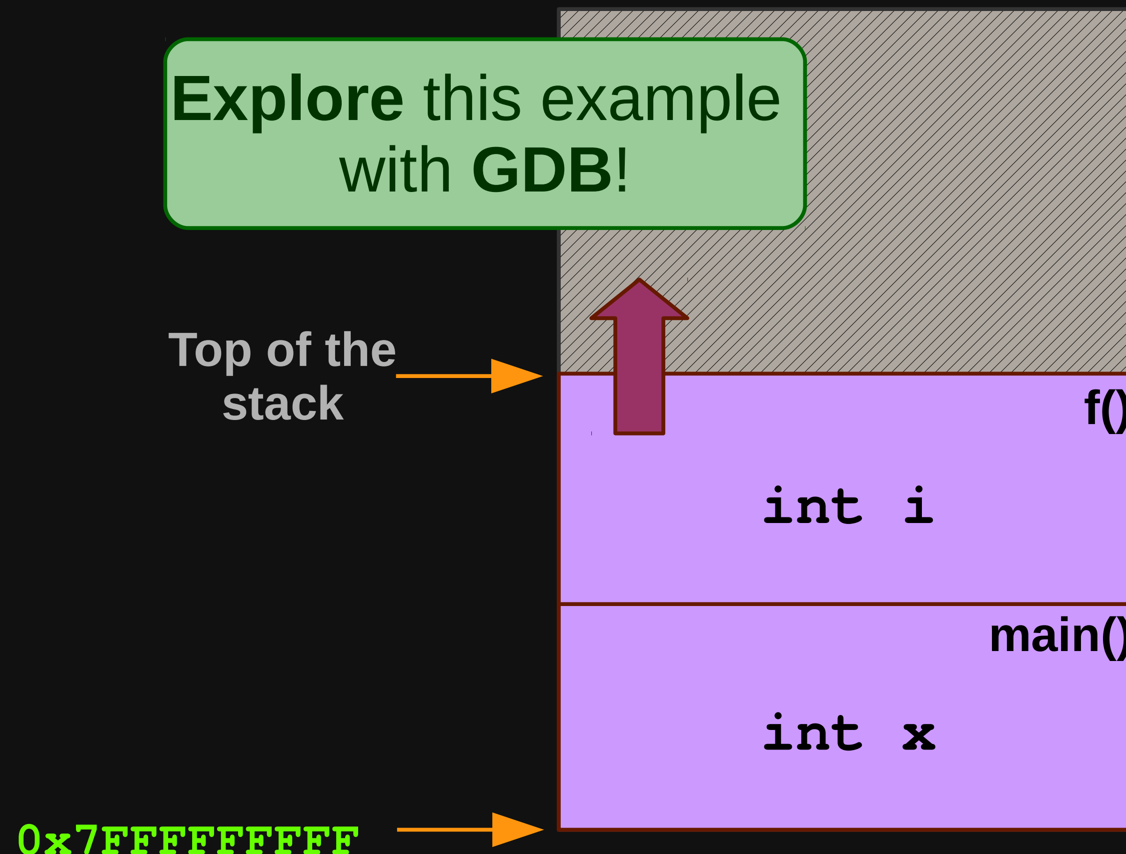
  Everything passed to functions are copies, including pointers!

- Example

byvalue.c

```c
#include <stdio.h>

void f( int i ) {
    i= i + 1;
    printf( "value: %i\n", i );
}

int main() {
    int x = 10;
    f( x );
    printf( "value: %i\n", x );

    return 0;
}
```

Explore this example with **GDB**!



Parameter Passing

- And how about **by-reference** parameter passing?

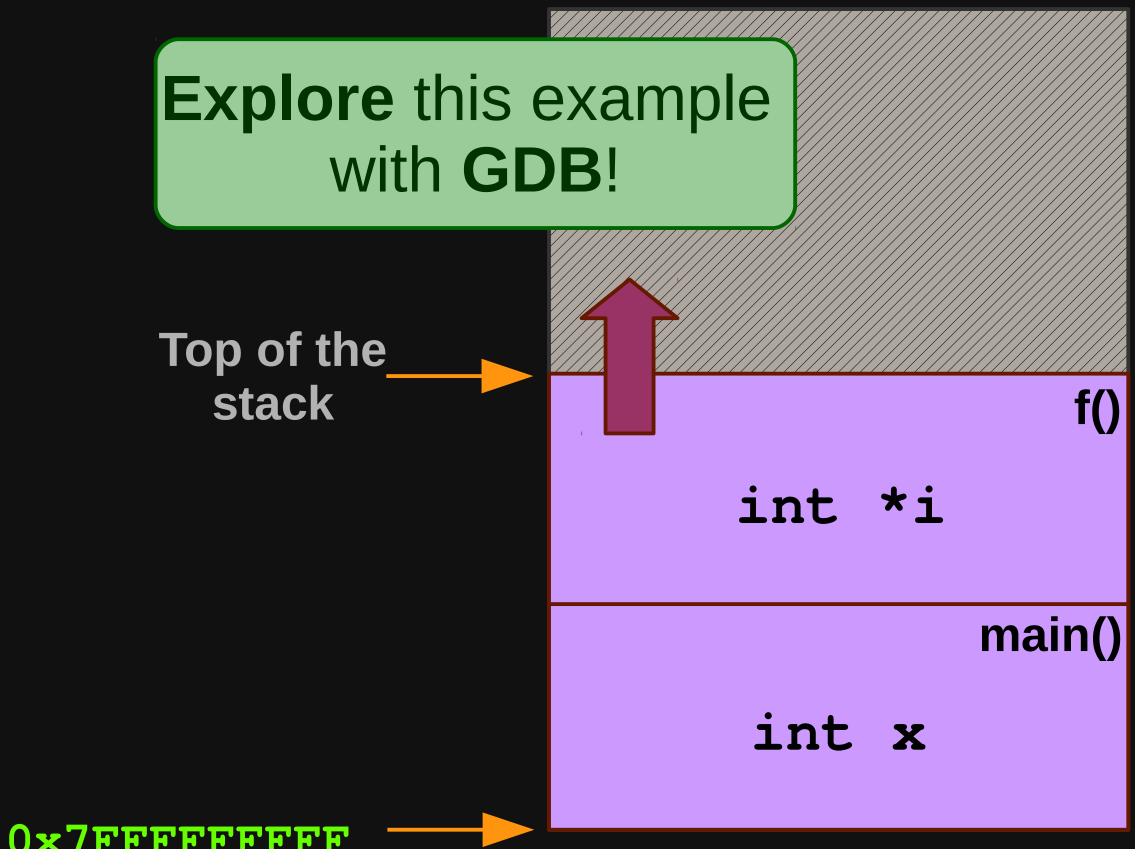
  Actually, it is by−value!!

- Example

Explore this example with **GDB**!

byreference.c

```
#include <stdio.h>

void f( int *i ) {
    *i= *i + 1;
    printf( "value: %i\n", *i );
}

int main() {
    int x = 10;
    f( &x );
    printf( "value: %i\n", x );

    return 0;
}
```



Changing a Pointer Parameter

Inserting a new node after the given one:

ptr2struct.c

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

struct Node {
    int value;
    struct Node *next;
};

void InsertAfter(struct Node *root, int value) {
    root->next = (struct Node*) malloc (sizeof(struct Node));
    root->next->value = value;
    root->next->next = NULL;
}

int main() {
    struct Node *n1 = (struct Node*) malloc (sizeof(struct Node));
    n1->value = 10;
    n1->next = NULL;
    InsertAfter(n1, 20);

    return 0;
}
```

Implement the function **`InsertBefore(...)`**!

→ Given a pointer to the first node, insert the new node before it. Update the pointer parameter to point to the new node!

DIY!

Changing a Pointer Parameter

Solution:

ptr2ptr2struct.c

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

struct Node {
    int value;
    struct Node *next;
};

void InsertBefore(struct Node **root, int value) {
    struct Node *tmp_ptr = (struct Node*) malloc (sizeof(struct Node));
    tmp_ptr->value = value;
    tmp_ptr->next = *root;
    *root = tmp_ptr;
}

int main() {
    struct Node *n1 = (struct Node*) malloc (sizeof(struct Node));
    n1->value = 10;
    n1->next = NULL;
    InsertBefore(n1, 20);

    return 0;
}
```

You will need a **pointer to a pointer** in order to **change** the pointers' **value**!

References

- **Understanding Memory**. University of Alberta, 2008.
- **Virtual Memory**. Wikipedia.
  - https://en.wikipedia.org/wiki/Virtual_memory
- **Memory**. Florent Bruneau. Intersec TechTalk. 2013.
  - https://techtalk.intersec.com/2013/07/memory-part-1-memory-types
- **Linux Memory Management System Source Code**
  - http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/mm

Further Reading

- **What Every Programmer Should Know About Memory**. Ulrich Drepper. Red Hat, Inc. 2007.