CSE 303
Concepts and Tools for Software Development

Magdalena Balazinska

Winter 2010

Lecture 12 – Structs and Heap

What We Have Seen So Far

Introduction to C

- Structure of a C program
- Memory model of a process
- Pointers and the stack
  - Pointers to basic data types
  - Arrays and strings
  - Passing arguments to functions (including pointers)
- Formatted input and output
  - Writing formatted data to stdout, stderr, or a file
  - Reading formatted data from stdin or from a file

Tools: debugger and version control system

Where We Are Going This Week

- Defining new data types
  - Structures in C
  - Converting between types: typecasts

- Dynamic memory management
  - The heap
  - Building, maintaining, destroying data structures
    - Example: lists, queues, trees

Structure Definition

- A structure is a "collection of related variables under one name"

```
struct sensor_reading {
  long timestamp;
  char location[20];
  int temperature;
};
```

  - The related variables can be of different types
  - So a structure is basically a record
  - Often a **building block** for more complex data structures: linked lists, trees, queues, etc.

Structure Variables

- Method 1 to declare structure variables

```
The code on the previous slide followed by
struct sensor_reading v;
struct sensor_reading a[2];
struct sensor_reading *p;
```

- Method 2 to declare structure variables

```
struct sensor_reading {
  long timestamp;
  char location[20];
  int temperature;
} v, a[2], *p;
```

Structure Variables

- Method 3 to declare structure variables

```
typedef struct sensor_reading Reading;
Reading v, a[2], *p;
```

- Keyword `typedef` serves to define synonyms (aliases)
- Creating the structure and type in one statement

```
typedef struct {
   long timestamp;
   char location[20];
   int temperature;
} Reading;
```

Using structs

Initializing: `Reading v = {1002,"EE037",67};`

Accessing fields

```
v.timestamp = 1002;
Reading *p = &v;
(*p).timestamp = 1002;
```

Shorthand notation: `p->timestamp = 1002;`

- **Reminder**: When passing a struct to a function as argument, we will pass a copy of that struct (called "passing by value")

Examples: `struct.c, struct-functions.c`

Types in C

- There are an infinite number of types in C, but only a few ways to create them:
  - `char`, `int`, `double`, etc.
  - `void` (no data type, absence of data type)
  - `struct T`
  - arrays
  - `t*`, where `t` is a type
  - `union`, `enum` (not covered, read on your own)
  - function pointers (extra credit question on hw4)
  - `typedefs` (just expand to their definitions)

Unary Type Cast Operator

- Goal
  - Convert an expression from one type to another
- Syntax: `(t)e`
  - Where `t` is a type and `e` is an expression
- Examples

```
int a=3; float b=4.3; long l=LONG_MAX;
printf("%d %f ",(int)b,(float)a);
printf("%ld %hu",l,(unsigned short)l);
```

- Output: `4 3.000000 2147483647 65535`

Casts Semantics

- Semantics depend on what you are casting
- Casting between numeric types
  - To wider type, get same value
  - To narrower type, may not (will get mod)
  - From floating point to integer (will round)
- Casts are explicit conversions
- There are also a lot of implicit conversions
  - Example: `int a = 3.0 * 1;`
  - Other example are arguments in function calls

Casting Pointers

- If `e` has type `t1*`, `(t2*)e` is a pointer cast
  - After casting, still pointing to the same location in memory
- Example

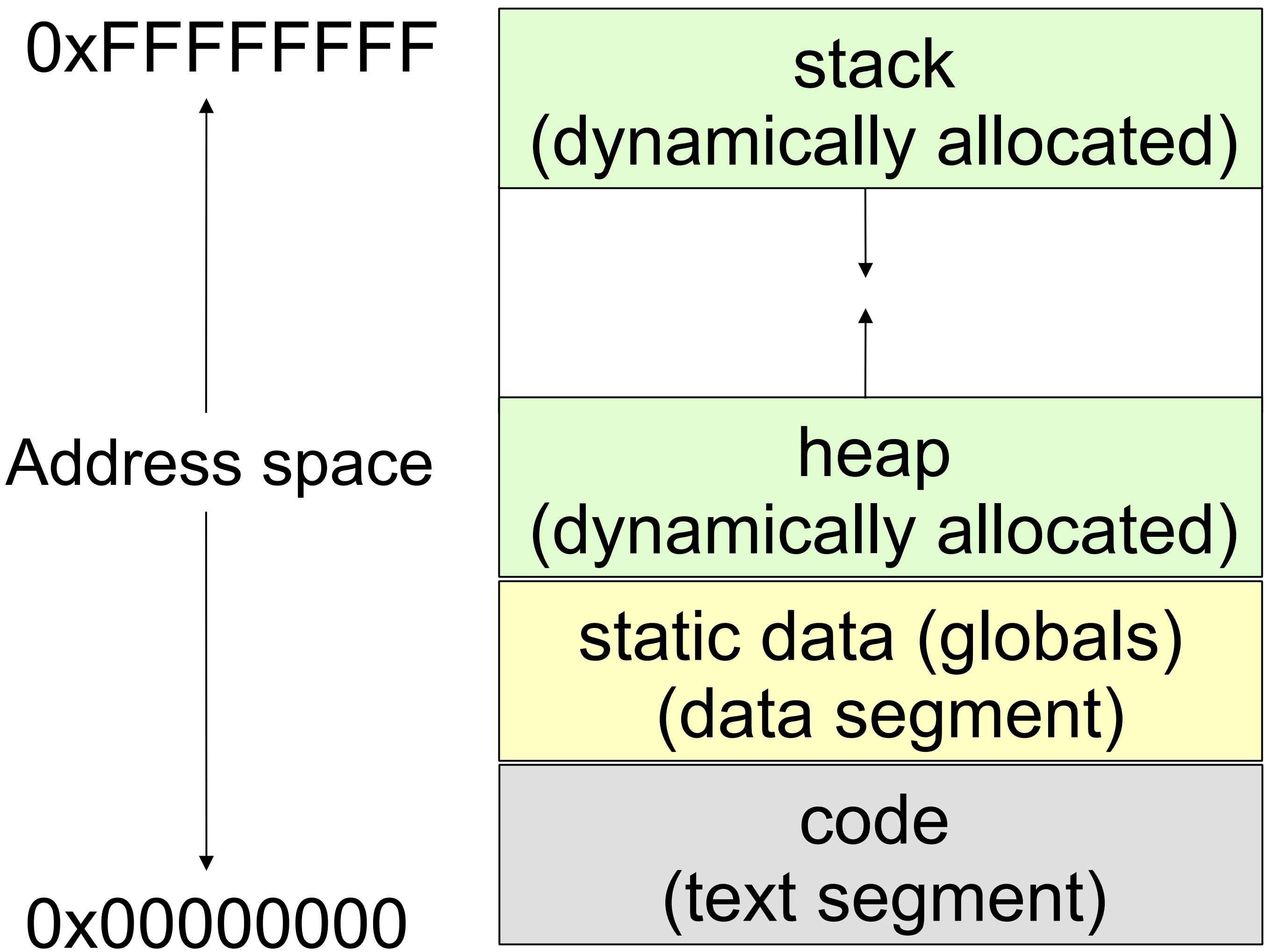
```
int array[10];
int *p1 = &array[1]; int *p2 = &array[2];
printf("%d ", p2 - p1);  // Output: 1
printf("%d", (char*)p2 - (char*)p1); // Output: 4
```

- Note: compiler will let you do what you want without checking
- Casts are thus unsafe and can set your computer on fire
- Examples: `cast.c`

Memory Management

- So far, space for all our variables was allocated on the stack (except for global variables)
- Problems
  - Space is reclaimed when allocating function returns
  - Variables have fixed size
- What if would like to
  - Allocate space and keep it between function calls
  - Create **data structures that grow & shrink with time**
- Solution: need to use the heap

Address Space of a Unix Process



Address space is just array of 8-bit bytes

Typical total size is: 2^{32}

We will assume that integer is 4 bytes

A *pointer* is just an index into this array

Dynamic Memory Management

`void* malloc(size_t size);`

- Allocates a chunk of memory on heap
- Returns pointer to chunk or NULL

`free(void* ptr);`

- De-allocates chunk of memory previously allocated with malloc

- Examples: `struct-dynamic.c`
- Note:
  - In Java `new C(...)` also uses the heap
  - Garbage collector takes care of freeing space

Simple Example

```
// Allocate a chunk of memory
Reading *p = (Reading*)malloc(sizeof(Reading));
// Check if allocation succeeded
if ( !p ) { ... }
// Initialize and use allocated chunk of memory
pointer->ts = 10;
pointer->temp = 70;
// Free the chunk of memory
free(pointer);
pointer = NULL;
```

Example 2: Growable Arrays

- Step 1: Dynamically-allocated array of size X

```
Reading *a=(Reading*)malloc(X*sizeof(Reading));
```

- Step 2: Growing the array
  - Step 2.1 Allocate a new, larger array
  - Step 2.2 Copy all elements
  - Step 2.3 Deallocate old array
- Example `growing-array.c`
- Further reading: `calloc and realloc`

Readings

- Programming in C
  - Chapter 9 "Working with Structures"
  - Chapter 14
    - Section on "Typedef Statement" (pp 325-327)
    - Section on "Data Type Conversions" (pp 327-330)
  - Chapter 17
    - Section on "Dynamic Memory Allocation" (pp 383-388)