Operating Systems Lab (C+Unix)

Enrico Bini

University of Turin

Outline

Scope of a variable

- The *scope* of a name (variable or function) is the portion of code where that name is visible and then it may be used
- The scope of a name (variable) declared within a function is restricted to that function
- Parameters of functions have the same scope of a variable declared inside a function
- The scope of a name declared outside any function (*global variables* or *function declarations*) is from the place of declaration until the end of the file
- If a variable with the same name is declared both outside and inside a function, the one inside the function prevails

Global variables

- Global variables are declared outside any function
- Global variables are visible to all functions from the declaration to the end of file
- Usage of global variables:
  - good: when many functions share a large amount of data, the usage of global variable is more efficient (it prevents parameter passing)
  - bad: the code relying much on global variable may be:
    1. hardly portable: functions are not really "isolated" pieces of code
    2. hard to comprehend/debug: when the reader finds a global variable, it may be not obvious where it is declared
- If a function uses a global variable as input or output, it is **strongly recommended** to add a comment on top of the function
- Names of global variables should be highly informative to avoid the reader to browse much code:
  - `number_students` is good
  - `n` is bad

Outline

Storage classes

- All C variables have a **storage class**, which determines where variables are stored
- Normally, variables are stored in memory. Three possible areas of memory:
  1. variables over the BSS (Block Standard by Symbol, historical acronym)
  2. variables over the stack segment
  3. variables over the heap
- Moreover,
  4. variables may be stored in registers
- Finally,
  5. the storage class may be delegated to other places in the code (*external varables*). More details in "Modular programming"

Memory segments

- Depending on the needs, the OS assigns a few memory segments to *processes* (which are programs in execution)
- Segments are mapped over the process address space
- Each segment has:
  1. start and end addresses (meaningful in the process address space)
  2. flags that determine the access modes:
     - ★ read: it can be read
     - ★ write: it can be written
     - ★ execute: it contains code which may be executed
     - ★ private/shared: if it isn't/is shared among other processes
- To view the memory mapping of a process, try:

```
ps -Af | grep sh
```
to get a Process ID (PID)

```
cat /proc/<PID>/maps
```
to print the memory map of <PID>

Outline

Variables on the BSS

- BSS is a read-write memory segment
- Size of BSS is decided at compile time (depending on the size of allocated variables, plus some padding for alignment)
- Two ways to allocate variables over the BSS
  1. global variables
  2. local variables declared with the `static` qualifier

```
void func(void) {
  static int my_static_var;
  ...
}
```

- Allocated: at the begin of the program
- Deallocated: at the end of the program

`static` variables within functions

- Scope: same as local variables (only within the function)
- Lifetime: same as global variables (from the start to the end of the program)
- Typical usage: to keep some state between consecutive invocation of a function

```
void func(void) {
  static int count_invocations = 0;

  ++count_invocations;
  ...
}
```

- Example:
  test-static.c

Outline

Variables on the stack

- When variables are declared at the top of a function, they are allocated onto the stack (unless the `static` qualifier is pre-fixed)
- Variable are allocated over the stack by reducing the stack pointer as needed by the size of the variables
- Allocated: when the function is entered
- Deallocated: when returning from the function
- Hence, we cannot rely on the their initial value
- The prefix `auto` in variables declaration, such as in

```
void func(void) {
  auto int my_stack_var;
  ...
}
```

  may be used. However, since it is the default allocation it is rarely (never?) used explicitly
- How much space is available on the stack?
  test-stack-killer.c

Content of the stack

- The stack is a memory area with LIFO (Last-In First-Out) policy
  - `push` assembly instruction stores data to top of the stack
  - `pop` assembly instruction extracts data from the top of the stack
- Main purpose is to store parameters and return address of function invocation
  - when a function is invoked (`call` assembly instruction),
    1. the parameters of the function invocations are pushed to the stack
    2. the return address is pushed to the stack
    3. then the control flow goes to the invoked function
  - when we return from function (`ret` assembly instruction))
    1. the return address is fetched from the stack
    2. the control goes back to the invoking function
- The example *test-stack.c*
  - shows the content of the stack (the parameters, the return address, etc.)
  - alters the content of the stack to modify the execution flow (typical attack)

Outline

Variables on the heap: dynamic allocation

- The heap (in Italiano "mucchio", "cumulo") is a memory area available to the program upon specific request to the operating system
- The program may ask the OS some memory via the following calls

```
#include <stdlib.h>

void * malloc(size_t size);
void * calloc(size_t nmemb, size_t size);
void * realloc(void *ptr, size_t size);
```

  which is returned via a pointer (`void *`)
  - `malloc` allocates `size` bytes in memory
  - `calloc` allocates `nmemb` elements of `size` bytes in memory and set them to zero
  - `realloc` changes the size of previously allocated area
- Allocating memory via `malloc` is called *dynamic memory allocation* because the size of the allocated memory is **decided at runtime**
- When variables are declared, the size of memory is decided at compile time (*static memory allocation*)

Standard ways for dynamic allocation

- Standard code to allocate an array of `num` elements

```
int * p;

p = malloc(num*sizeof(*p));
/* better than malloc(num*sizeof(int)) */
```

- `calloc(...)` has a slightly different syntax and it clears the memory (set all bytes equal to zero)

```
int * p;

p = calloc(num, sizeof(*p));
```

- After the allocation, the memory can be used as needed

```
p = calloc(num, sizeof(*p));
for (i=0; i<num; i++) {
  p[i] = i*i;   /* using array notation */
}
```

Memory must be freed

- All memory areas allocated by `malloc`, `calloc` and `realloc` must be released by `free`
- standard code to deallocate a memory area pointed by `p` is

```
free(p);
```

- `free(p)` is error, if `p` not returned by `malloc`/`calloc`
- A special care must be taken to free a memory area before the pointer to the area is lost

```
p = malloc(N*sizeof(*p));
...
p = &v;   /* ref to allocated mem is lost!! */
```

- To avoid forgetting to free the memory, it is recommended to write the `free` code immediately, possibly at the bottom of the file.
- Lifetime of memory allocated onto the heap
  - Allocated: when `malloc()`/`calloc()` is invoked
  - Deallocated: when `free()` is invoked (or at the end of the program)

Static vs. dynamic allocation

- Is it better static allocation

```
int v[100];
```

- or, dynamic allocation

```
int * v;
v = malloc(100*sizeof(*v));
```

- used by same syntax: `v[10] = 412;`
- Dynamic allocation can use less memory than static allocation (static allocation requires overallocating, by dynamic allocation memory can be allocated when needed)
- Static allocation is faster since it avoids expensive system calls such as `malloc` and `free`
- Example of usage of `malloc`
  test-malloc.c

Outline

Allocation of data in memory

- Let us have a look to the following examples:
  - *test-show-addr.c*
- Remember the difference between

```
char v[] = "string0";
char * p = "string1";
```

  - `string0` may be modified (it belongs to a page with "`w`" permission)
  - `string1` may not be modified (no "`w`" permission)

Outline

`register` variables

- The compiler may be informed that some variable **should be allocated to a register** of the processor by adding the keyword `register` at the declaration

```
register int my_register_var;
```

- `register` variables are used for frequently accessed variables: access time to a register if 10–100 times faster than access to memory
- The number of register is limited: the compiler cannot guarantee the allocation to a register