CS 240 Programming in C

Functions

February 27, 2024

main function

Functions

- Functions provide a convenient way to encapsulate some computation, which can then be used without worrying about its implementation
- The function name of its definition has to be unique in a C program.
- A C program starts execution with a function called main.

```
int main(int argc, char *argv[]){

}
```

- A C program can only have one copy of a main function definition.

Function Declaration and Definition

- Function declaration

```
int power(int m, int n);
```

  - Function declaration says that power is a function that expects two int arguments and returns an int

- Function definition

```
returnType funcName(parameter declarations, if any) {
  declarations

  statements
}
```

Parameters vs Arguments

- Formal parameter
  - The names were given to the arguments in the function definition
  - Often referred to as parameters
- Actual parameter
  - The names supplied in a function call
  - Often referred to as arguments

Call by Value vs Call by Reference

- In C all function arguments are passed "by value"
- The called function is given the values of its arguments in temporary variables, distinct from the originals
- This means that anything you do to a variable inside a function has no effect on that variable outside of the function
- When call by reference is used, we can alter an argument outside of the function with actions inside the function

Called by Value

Write a function to swap two integers.
Try it now.

Called by Value for array

When an array variable is passed into a function as an argument, what has been passed?
and what this means for manipulation of the array in local function to the array in the called scope.

Called by Value for array

```c
#include <stdio.h>

void f1(int nums1[]) {
    int nums2[10];
    printf("sizeof :%lu\n", sizeof(nums1));
}

int main() {
    int nums[10];
    printf("sizeof :%lu\n", sizeof(nums));
    f1(nums);
}
```

Listing 1: Array Parameter

Function Memory Layout

Layout

- Function calling stack grows from a high address to a lower address which means the inner local variable will have a lower address than the caller function scope
- Array index grows from a lower address to a higher address for a local variable

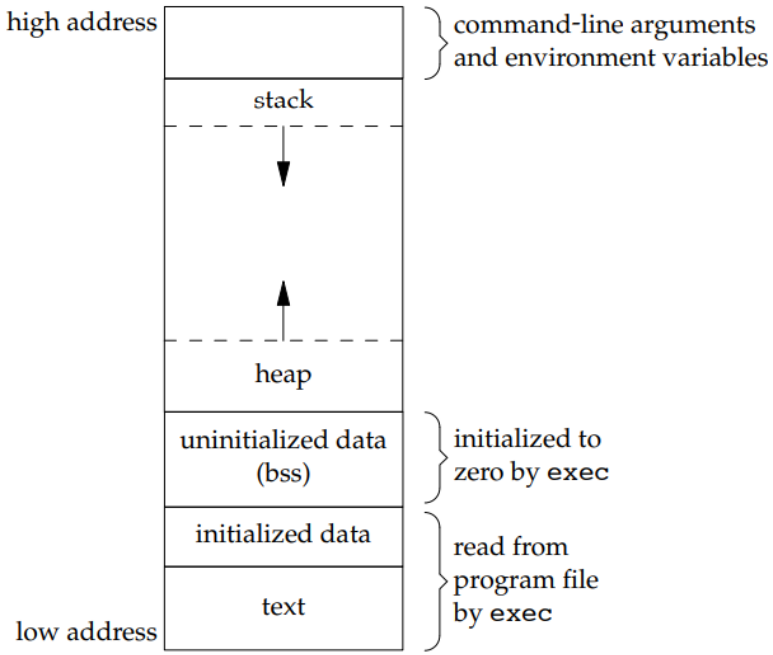


Figure: Function Memory Layout

Function Stack Memory Layout

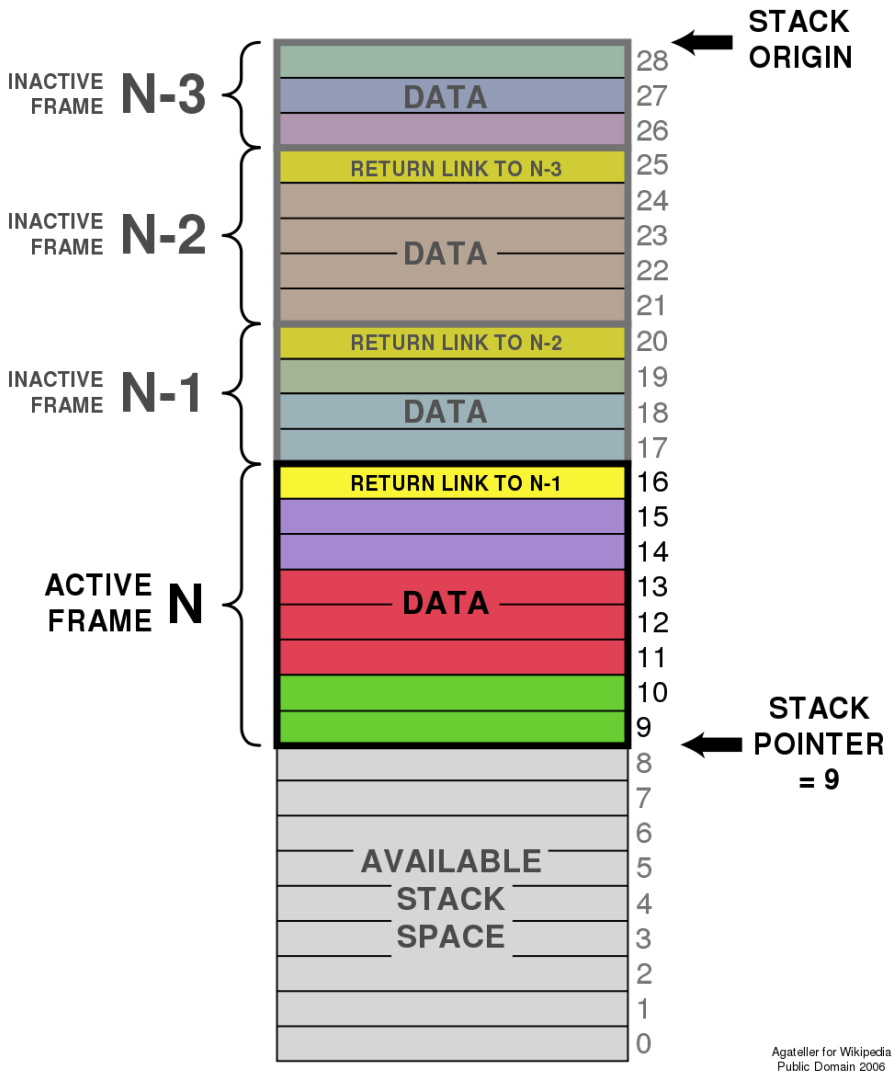


Figure: Stack Memory Layout

Memory layout for function call

```
1  //
2  //
3  //
4  #include "stdio.h"
5
6  int factorial(int a) {
7      printf(" factorial: a:%3d\taddress: %p\n", a, &a);
8      if (a == 1) return 1;
9      return a * factorial(a - 1);
10 }
11
12
13 int main(int argc, char *argv[]) {
14
15     int a = 10;
16     printf(" main: a:%d\taddress:%p\n", a, &a);
17     printf(" factorial of %d is: %d\n", a, factorial(a));
18     return 0;
19 }
```

Listing 2: Stack (Buffer) OverFlow Corruption

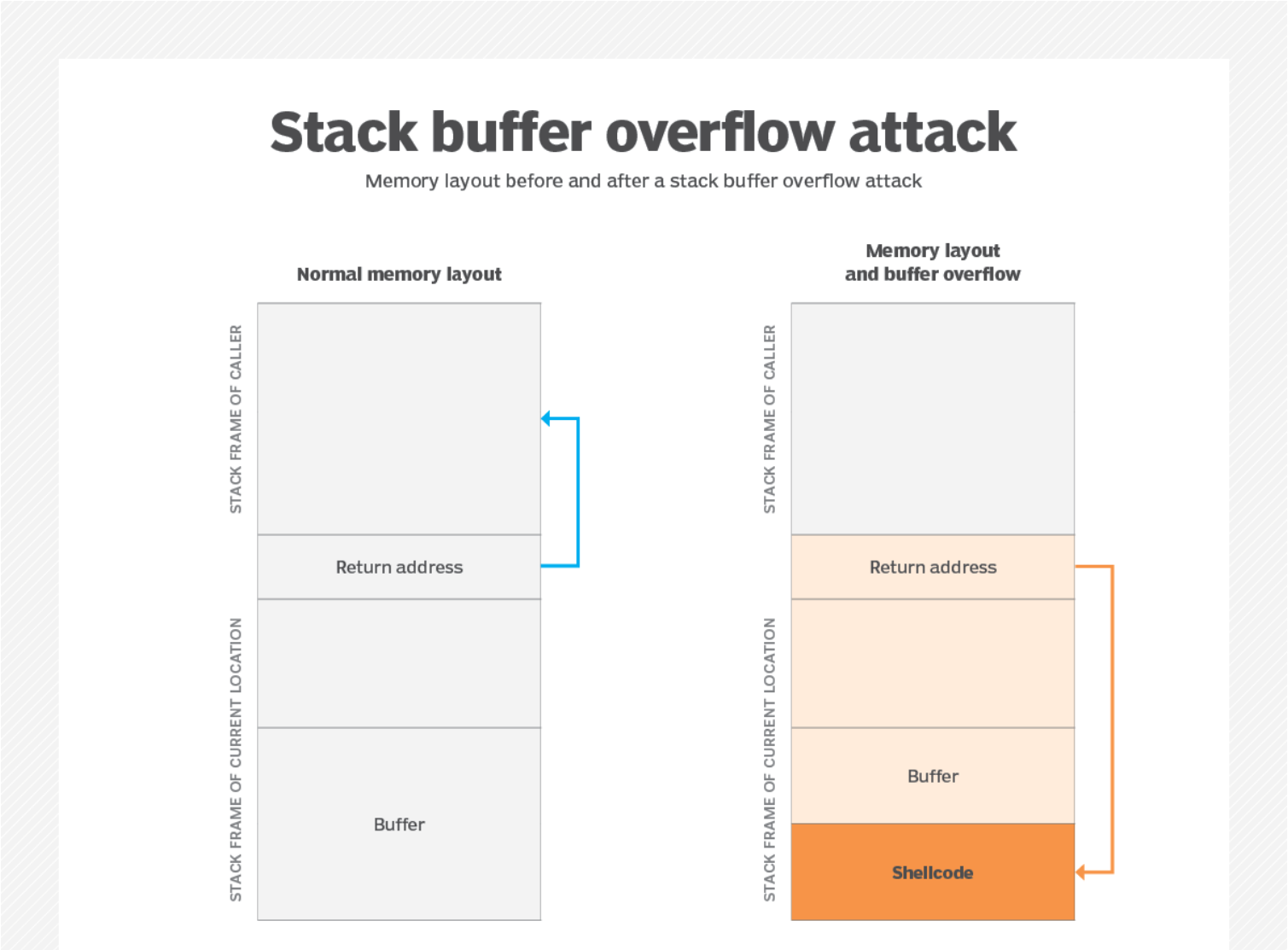
Memory Layout for array

```
1  //
2  //
3  //
4  #include <stdio.h>
5
6  void func(char nums[]) {
7      char nums_local[2] = {'c', 'd'};
8      for (int i = 0; 1; ++i) {
9          printf("main nums and local nums - %d addresses:"
10                 " %p\t%p\n", i, nums, nums_local + i);
11         if (nums_local + i == nums) {
12             printf(" Get to the main nums address from local"
13                    "the char is %c\n", nums_local[i]);
14             printf(" Change them to x and y\n");
15             nums_local[i++] = 'x';
16             nums_local[i++] = 'y';
17             return;
18         }
19     }
20 }
21
22 int main(int argc, char *argv[]) {
23     char nums[2] = {'a', 'b'};
24     func(nums);
25     printf(" From main: nums are: %c, %c\n", nums[0], nums[1]);
26 }
```

Listing 3: Stack (Buffer) OverFlow Corruption

Stack (Buffer) OverFlow

https://en.wikipedia.org/wiki/Stack_buffer_overflow
https://www.techtarget.com/whatis/definition/stack-overflow



Memory Layout

```
1  //
2  // Created by haoyu on 3/9/2023.
3  //
4  #include <stdio.h>
5
6  void func(char nums[]) {
7      char nums_local[2] = {'c', 'd'};
8      for (int i = 0; 1; ++i) {
9          printf("main nums and local nums - %d addresses:"
10                 " %p\t%p\n", i, nums, nums_local + i);
11         if (nums_local + i == nums) {
12             printf(" Get to the main nums address from local"
13                    "the char is %c\n", nums_local[i]);
14             printf(" Change them to x and y\n");
15             nums_local[i++] = 'x';
16             nums_local[i++] = 'y';
17             return;
18         }
19     }
20 }
21
22 int main(int argc, char *argv[]) {
23     char nums[2] = {'a', 'b'};
24     func(nums);
25     printf(" From main: nums are: %c, %c\n", nums[0], nums[1]);
26 }
```

Listing 4: Stack (Buffer) OverFlow Corruption

Segmentation Fault

Go beyond the memory boundary

- A segmentation fault occurs when your program attempts to access (mostly write to) an area of memory that it is not allowed to access.
- In other words, when your program tries to access/modify the memory that is beyond the limits that the operating system allocated for your program.
- This often happens in a C program, because the compiler will not check if your memory access would go out of the boundary, if you are modifying a memory that is read-only or not allocated yet, etc.

Example

Seg faults are mostly caused by pointers that are

- used to be properly initialized.
- used after the memory they point to has been reallocated or freed.
- used in an indexed array where the index is outside of the array

Segmentation Fault

Example

```
// write into constant memory cause a seg fault
// one of the examples to create a seg fault
char *s = "hello world";
s[1] = 1;
```

How to solve

- When seg fault happens, usually there isn't much useful information being printed out to the console like running a Java program.
- You can use a debugger to locate a run-time failure for the same condition and analyze it.
- You can also create a run time dump file which is a record of the state of the running program and analyze the dump file. But this is advanced stuff.
- For this class, we introduced how to use CLion debug to locate the error like how to set breakpoints, conditional breakpoints, and step next/in/out. It is our recommended way.

Block and Scope

- Block: A section of code that is grouped together
  - In C, blocks are delimited by curly braces

    ```
    { [block statements] }
    ```

  - or the parenthesis of for loop

    ```
    for (int i=0 ; i<10; i++);
    for (int i=0 ; i<10; i++) {int j = 0; j = 1};
    ```

- Scope: the area of a program where a variable can be referenced
  - For each different entity that an identifier designates, the identifier is visible (i.e., can be used) only within a region of program text called its *scope*

Block and Internal Variable

- Variables defined within a block are local to the block where they are defined which means that they are not accessible the outside of the block; they come and go with the block of codes executing and finishing.
- Internal variables are also often called "auto" variables. Inside a function block, these two definitions are equal (it is just the "auto" keyword is often ignored):

```
int j = 0;
auto int j;
```

- If a variable was not defined within this block, then it will resort to the outer block for the definition of this variable, until outside the function within the same file.

External/global

- For any variable defined outside of a function, it has a global/external scope.
- Because C does not allow function definition in another function, thus all functions are external/ global
- Because of that, function name has to be unique, even if they are compiled separately from different files since they will be used to reference their binary code body.
- Variables usually exist inside a function body, and for the same level of scope, there can not exist two variables that assume the same name; inside different functions, there can be variables that assume the same name.
- Like function names, external variable names must also be unique from each other.

Extern

Declaration of variable

- Remember to use a function or variable we will have to at least have a declaration, and the definition can reside in a third part library.
- Function declaration:

```
# This is a function declaration
# Since it has no body definition
int add(int a, int b);
```

- What about variable declaration only:

```
# This is the definition
int a;
```

- extern keyword

```
# This is the variable declaration
extern int a;
```

Extern

Declaration of variable

- Interesting fact that these two are equivalent. But since the function body can be used to differentiate the declaration and definition, then the extern keyword is redundant.

```
int add(int a, int b);
extern int add(int a, int b);
```

- For accessing an external variable that is not defined within this source code, the variable declaration can be useful.

Example

```
extern int i; // only this is declaration
extern int i=0; //This is Definition, not Declaration

int i; int i=0; // These are all variable definitions
```

A Summary

Question:

Are external variables the variables defined by the "extern" keyword?

External Variables and the "extern" key word

- No.
- An external variable is just a variable being defined outside any function.
- The "extern" keyword is used for searching the external/global variable reference somewhere else. It means there is no variable definition here within this function.

Declaration vs Definition

- A variable definition is also a declaration, but a declaration is not necessary to be a definition.
- A variable is to be used, has to at least have a declaration first.

External Variables

Advantages:

- If a large number of variables must be shared among functions, external variables are more convenient and efficient than long argument lists.
- External variables also retain their values after the exit of a function call, since no function owns it solely.
- External (global) variables are favored in high-performance computing. They allow additional optimization by compilers.

External Variables

Disadvantages:

- It is problematic for decoupling a program structure, which makes a big software into less dependent parts such that it is easy for maintaining and test etc.
- If their value gets corrupted, hard to trace the reason. They make functions dependent on their external environment
- In fact, software architecture/design standards often prohibit the use of external variables

External Variables (External Static)

- External variables can be accessed by any function in the program.
- what if we want to limit its scope?
- The static declaration applied to an external variable or function limits the scope of that object to the source file being compiled.

Static Local Variable

- Static local variable is a local variable that retains and stores its value between function calls or blocks and remains visible only to the function or block in which it is defined.

Example: Static External

```
#include <stdlib.h>
double drand48(void);
void srand48(long int seedval);
```

- The pseudorandom number generator `drand48()` is a family of functions
- They keep an external static variable X as the seed of the generators
- We must call `srand48()` to initialize the seed to generate a different sequence of numbers.

Example: Static Internal

```
#include <stdio.h>
int counter(){
     static int num;
     return num++;
}

int main(void){
     for (int i=0;i<5;i++) counter();
     printf("%d\n", counter());
}
```

The `register` Variables

- A register declaration advises the compiler that this variable will be heavily used
- We want it placed in a machine register, but the compiler is free to ignore this suggestion if it needs registers
- Can only be applied to automatic variables

The `register` Variables

- Register variables can be defined as local variables within functions or blocks, they are stored in CPU registers instead of RAM to have quick access to these variables.

  `Example: register int age;`
- A register variable may actually not be placed into registers in many situations.
- And it is not possible to parse the address of a register variable regardless of whether the variable is actually placed in a register.
- The specific restrictions on the number and types of register variables vary from machine to machine.

Example: Register Variables

The variables declared using the register have no default value. These variables are often declared at the beginning of a program.

```
#include <stdio.h>

int main(void) {
{
    register int  i;
    int *p=&i ;
    /*it produces an error when the compilation occurs,
    we cannot get a memory location when dealing
    with CPU register*/

    return 0;
}
```

Summary

Storage Class	Declaration	Storage	Default Initial Value	Scope	Lifetime
auto	Inside a function/block	Memory	Unpredictable	Within the function/block	Within the function/block
register	Inside a function/block	CPU Registers	Garbage	Within the function/block	Within the function/block
extern	Outside all functions	Memory	Zero	Entire the file and other files where the variable is declared as extern	program runtime
Static (local)	Inside a function/block	Memory	Zero	Within the function/block	program runtime
Static (global)	Outside all functions	Memory	Zero	Global	program runtime