C PROGRAMMING

Ruben Acuña

Calvin Cheng

Fall 2017

2

DEVELOPMENT IN A VIRTUAL MACHINE

OVERVIEW

- In this course, we will want to develop software that not just runs on Linux but also that interacts with the operating systems.
- This is a potential issue since someone may not have Linux installed, or may be concerned that something they do will break the OS.
- Our solution will be to do development in a *virtual machine* (VM): a virtualized computer upon which we can install any operating system that we want. Typically some number of client OSs run on a host.
  - *Host*: This is the top-level computer which is using some OS and which has VM software (e.g., VirtualBox) installed on it.
  - *Client*: This is the OS (e.g., Xubuntu) installed onto a virtual machine when is running in the VM software.
- For the purpose of this course, we will use *Xubuntu*. Xubuntu is version of the popular *Ubuntu* Linux distribution which uses a light-weight GUI interface called *Xfce*.
- We will do our C programming under Xubuntu.

A SIMPLE C PROGRAM

THE MAIN() FUNCTION

- Every C program contains a single function called main which acts as the entry point to the program.
- Unlike other languages, the main is not contained by any class.

The simplest programs are thus:

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

What's this?

Certain parameters can be added to the main() function to allow for command line inputs:

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

SAMPLE C PROGRAM (EXAMPLE-2-01.C)

```
/* In this program, we ask the user to enter a number from the keyboard,
process the number and then print the result. */

#include <stdio.h>

#pragma warning(disable: 4996)       // ANSI C has been deprecated in VS2012
                                     // Add this line to remove error msg
int n = 5;

int main() {
     int i;

     printf(“Enter an integer: ");
     scanf("%d", &i);

     if (i > n)
          n = n + i;
     else
          n = n - i;

     printf("i = %d, n = %d\n", i, n);

     return 0;
}
```

FORMATTED C-STYLE INPUT/OUTPUT

Huh?

- Input: scanf (control sequence, &variable_1, … &variable_n);
    - &variable: address of the variable
- Output: printf (control sequence, expressions);
- The "expressions" is the list of expressions whose values are to be printed out. Each expression is separated by a comma.
- The control sequence includes the constant string to be printed, e.g., "i = ", and control symbols to be used to convert the input/output from their numeric values that are stored in the computer to their character format displayed:

%d	for integer
%f	for floating point
%c	for character
%s	for string of characters

SAMPLE C PROGRAM (EXAMPLE-2-02.C)

```
/* In this program, we ask the user to enter a number from the keyboard and
then print "Hello World" n number of times. */

#include <stdio.h>

#pragma warning(disable: 4996)

int main() {
     int i, n;

     printf("Hi, please enter an integer: ");
     scanf("%d", &i);

     for (n = 0; n < i; n++) {
          printf("%d - Hello World!\n", n + 1);
     }

     return 0;
}
```

Looks a little off…

C SCOPE RULES

Pretty standard: variables and functions must be declared before they are used.

```
void main() {
        int height = 6;  int width = 6;
        int area = height * width;
}
```

However, in C we also need to worry about *forward declarations* (aka prototypes), which make a name known before it is used. Since C uses only pass (to bottom) to compile a file, it needs to know about each function before it is called by another function.

In languages like Java that use a *two pass compiler*, this isn't necessary.

FORWARD DECLARATION

```
void bar(void);  // forward declaration to satisfy scope rule
int foo(void);   // forward declare all functions

. . .

int foo(void) {              // genuine declaration
    . . .
    bar();                   // call function bar()
    . . .
}

void bar(void) {             // genuine declaration
    . . .
    k = foo();               // call function foo()
    . . .
}
```

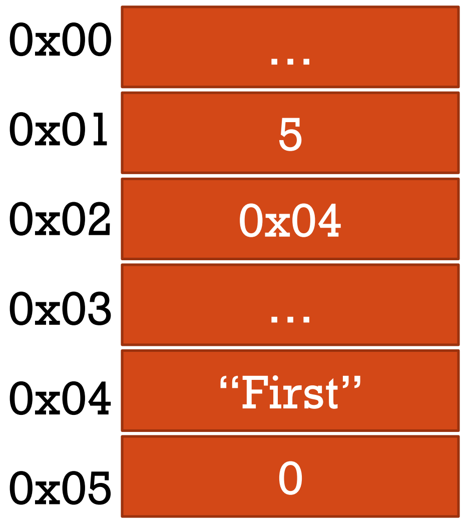
MEMORY IN C

MEMORY MANAGEMENT

Operating system memory management: allocate a piece of memory to each task (job). The memory allocated to a program is divided into three areas:

- Static Memory
- Stack
- Heap

All of these have some "physical" representation.



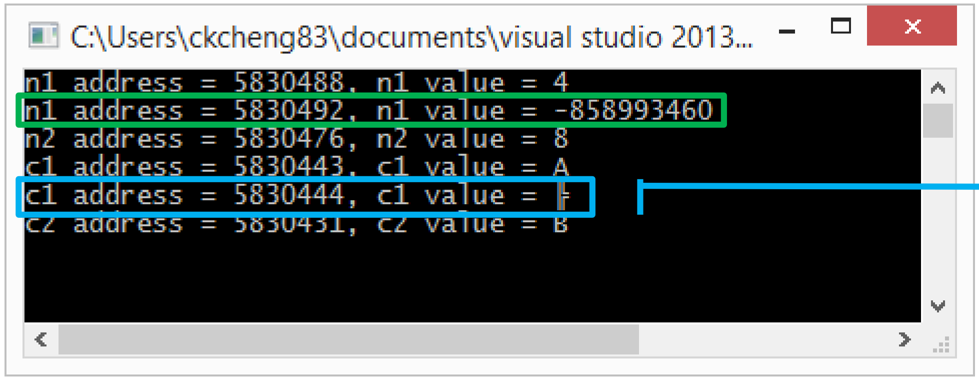
BYTE ADDRESSABLE MEMORY (EXAMPLE-2-03.C)

Byte addressing refers to hardware architectures, which support accessing individual bytes of data rather than words. It allows the system quick access to data within system memory.

```c
#include <stdio.h>

void main() {
     int n1 = 4, n2 = 8;
     int *pn1 = &n1, *pn2 = &n2;
     char c1 = 65, c2 = 66;
     char *pc1 = &c1, *pc2 = &c2;
     printf("n1 address = %d, n1 value = %d\n", pn1, *pn1);
     pn1 = pn1 + 1;
     printf("n1 address = %d, n1 value = %d\n", pn1, *pn1);
     printf("n2 address = %d, n2 value = %d\n", pn2, *pn2);
     printf("c1 address = %d, c1 value = %c\n", pc1, *pc1);
     pc1 = pc1 + 1;
     printf("c1 address = %d, c1 value = %c\n", pc1, *pc1);
     printf("c2 address = %d, c2 value = %c\n", pc2, *pc2);
}
```

BYTE ADDRESSABLE MEMORY (EXAMPLE-2-03.C)

```c
#include <stdio.h>

void main() {
	int n1 = 4, n2 = 8;
	int *pn1 = &n1, *pn2 = &n2;
	char c1 = 65, c2 = 66;
	char *pc1 = &c1, *pc2 = &c2;
	printf("n1 address = %d, n1 value = %d\n", pn1, *pn1);
	pn1 = pn1 + 1;
	printf("n1 address = %d, n1 value = %d\n", pn1, *pn1);
	printf("n2 address = %d, n2 value = %d\n", pn2, *pn2);
	printf("c1 address = %d, c1 value = %c\n", pc1, *pc1);
	pc1 = pc1 + 1;
	printf("c1 address = %d, c1 value = %c\n", pc1, *pc1);
	printf("c2 address = %d, c2 value = %c\n", pc2, *pc2);
}
```



```
n1 address = 5830488, n1 value = 4
n1 address = 5830492, n1 value = -858993460
n2 address = 5830476, n2 value = 8
c1 address = 5830443, c1 value = A
c1 address = 5830444, c1 value = ╠
c2 address = 5830431, c2 value = B
```

15

C-STYLE STRINGS

ARRAY AND STRING

There are two methods to initialize an array of characters in declaration:

```
char s1[ ] = {'a', 'l', 'p', 'h', 'a'};        // as an array of char
char s2[ ] = "alpha";                          // as a string
```

However, both will return different results in memory:

s1	a	l	p	h	a

s2	a	l	p	h	a	/0

where '\0' is the *null character*, marking the end of a string. It is automatically added to the end of a string. To achieve the same result in s1, we can add the null character afterward.

Truncated initialization: If the size of the array is specified and there is not enough space, the null character will not be attached to the string.

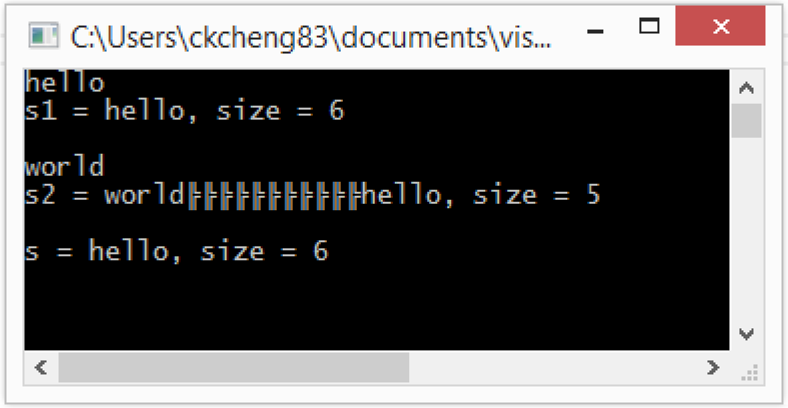
```
char s3[5] = "alpha";
char s4[4] = "alpha";
```

s3	a	l	p	h	a
s4	a	l	p	h	

STRING EXAMPLE (EXAMPLE-2-04.C)

Consider what the output of the following code might be:

```c
#include <stdio.h>
#include <string.h>   // strcpy

#pragma warning(disable: 4996)

void main() {
     int i;
     char s[6], s1[] = "hello";
     char s2[5] = "world";

     for (i = 0; i < 5; i++) { printf("%c", s1[i]); }
     printf("\ns1 = %s, size = %d\n\n", s1, sizeof(s1));

     for (i = 0; i < 5; i++) { printf("%c", s2[i]); }
     printf("\ns2 = %s, size = %d\n", s2, sizeof(s2));

     strcpy(s, sizeof(s), s1);
     printf("\ns = %s, size = %d\n", s, sizeof(s));
}
```



```
hello
s1 = hello, size = 6

world
s2 = world╠╠╠╠╠╠╠╠╠╠╠hello, size = 5

s = hello, size = 6
```

18 POINTERS

POINTERS

- Recall, variable names are simply binding a physical memory address to a human readable symbol (name).
- Although we are use to manipulating symbols to obtain data, we can also use the underlying addresses which have some structure.
- Direct manipulation of addresses can be useful.

If we use a variable to store an address, it is called **pointer type**.

Careful: all variables store a *value* at some address. It's just that sometimes that *value* at that address *is itself an address*.

POINTERS

Would it make sense to be able to assign a generic integer into a pointer?

Operations on pointer variables:

- An address value can be assigned to a pointer variable.
- The address stored in a point variable can be modified.
- **Referencing**: Obtain the address of a variable from its name.
- **Dereferencing**: Returns the value pointed by a pointer.

This is not a logical and!

C has unary 2 pointer operations: & and *

&	is a referencing operator that returns the address value of the variable it precedes. If y = &x;, then &x is called a "r-value".
*	returns the value at the address stored by the variable it precedes. If *y = 5;, then *y is called a "l-value" (locator value).

They are complementary:

* converts pointer to address	&(*p)	→	p
& converts address to variable content	*(&x)	→	x

POINTER EXAMPLE 1

```
int y = 10;
int* yptr;
int *xptr;        // Is there a difference between int* vs. *xptr?

yptr = &y;
xptr = &y;        // Is this legal?

*yptr = 20;       // Dereferencing?
*xptr = 30;       // What is the value of y?  10 or 20 or 30
```

Notice that an asterisk is used to denote a pointer type, but it is also used for dereferencing.

POINTER EXAMPLE 2

In this example, assume the value of x is 0, the address of x is 2000, and the address of y is 2004.

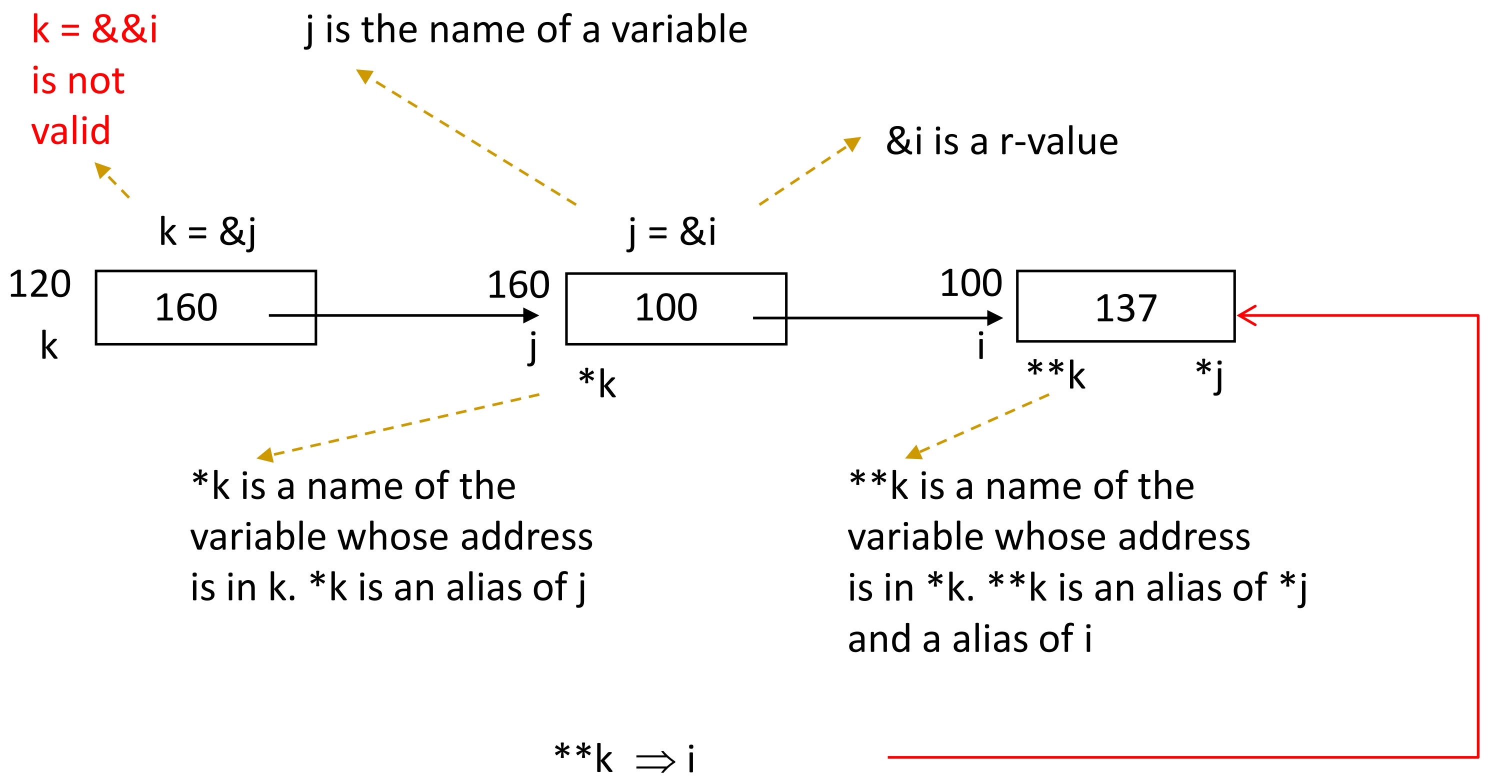
```
1:     int x, *y;

2:     y = &x;

3:     *y = *y + 10;

4:     y++;
```

After Line	x	y
2		
3		
4		

DOUBLE POINTERS

```
int i = 137, *j, **k;
j = &i;
k= &j;
**k = 0;
```

What happens if you declare *k, instead?

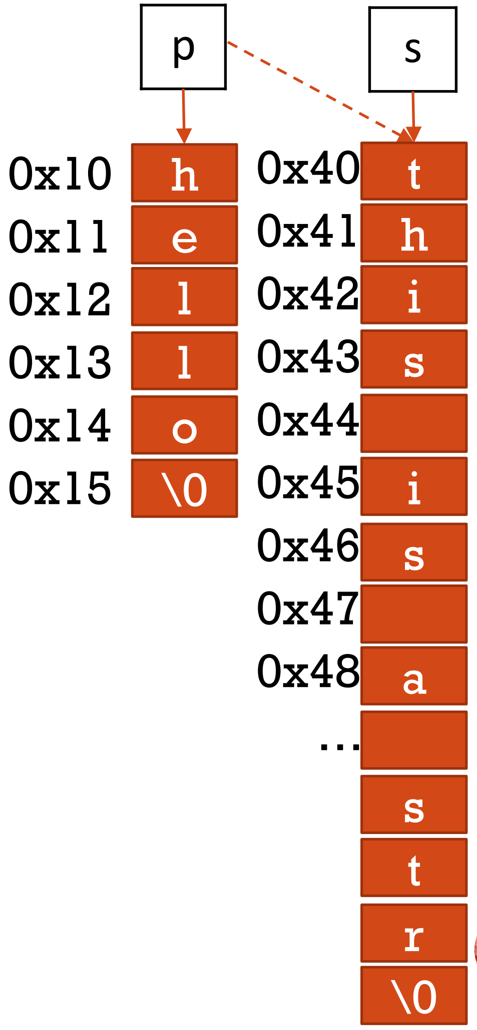


k = &&i is not valid
j is the name of a variable
&i is a r-value
k = &j
j = &i
120
160
k
160
100
j
*k
100
137
i
**k
*j
*k is a name of the variable whose address is in k. *k is an alias of j
**k is a name of the variable whose address is in *k. **k is an alias of *j and a alias of i
**k ⇒ i

POINTERS DECLARATION AND APPLICATION

Although a pointer can point to a variable of any type, it is especially common for complex/compound types. Why?

```
void main() {
    char *p = "hello", *s = "this is a str";
    p = s;
    printf("%s\n", p);
}
```



POINTERS AND MULTI-DIMENSION ARRAY (EXAMPLE-2-08.C)

Depending on your programming language, 2D arrays will be stored in different formats. C uses *row-major* order, where rows are stored one after another.

```c
#include<stdio.h>
void main() {
     char *p = 0;
     char ma[2][4];      //   0123
     ma[0][0] = ‘a';     //0: abcd
     ma[0][1] = ‘b';     //1: efg\
     ma[0][2] = ‘c';
     ma[0][3] = ‘d';
     ma[1][0] = ‘e';
     ma[1][1] = ‘f';
     ma[1][2] = ‘g';
     ma[1][3] = '\0';
     p = ma;
     while (*p) {
          printf("%c", *p);
          *p = *p + 1;       // Cheapo Encryption
          p++;
     }
     printf("\n");
}
```

Memory

ma[0][0]	a
ma[0][1]	b
ma[0][2]	c
ma[0][3]	d
ma[1][0]	e
ma[1][1]	f
ma[1][2]	g
ma[1][3]	\0

MIS-USING POINTERS EXAMPLE (EXAMPLE-2-09.C)

C allows constants to be declared using the *const* keyword…

```
#include <stdio.h>

void main() {
    const int i = 5;
    int *j;

    // i = i + 2;                // What would happen? Why?

    j = &i;
    printf("i = %d\n", i);

    *j = *j + 2;                 // What would happen? Why?

    printf("i = %d\n", i);       return 0;
}
```

CONSTANTS IN C

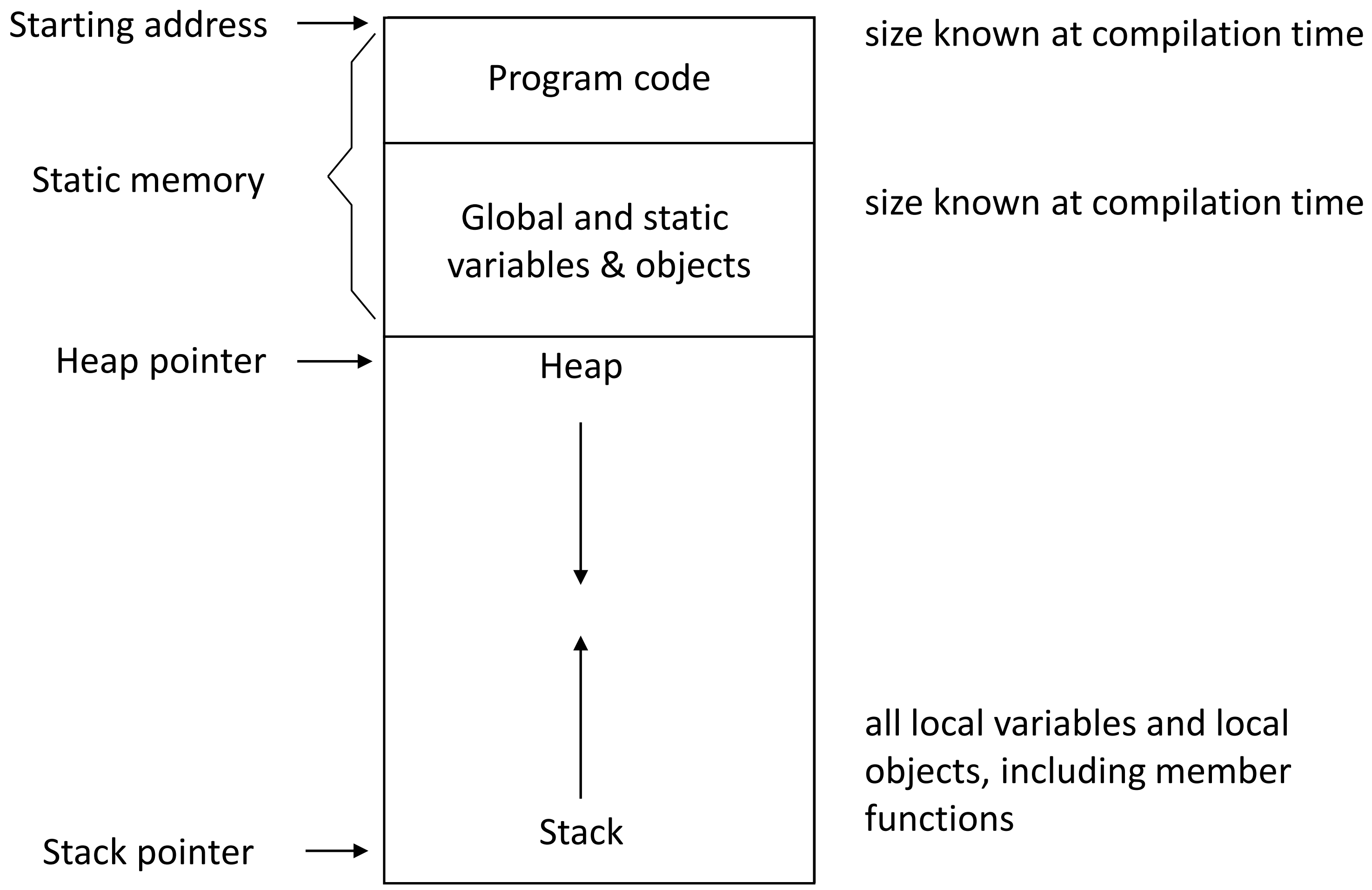
There are two mechanisms for representing constants:

- (macros) The compiler may at compilation time substitute values for constant definitions -- faster execution, may fit in one instruction. The constant is defined by a macro.
- (const) A constant may be a 'variable' which the program cannot modify -- have to read memory: slower.
  - A constant defined by *const* is actually a full variable:
    - It has memory allocated
    - You can use the ampersand (&) function to find its address and indirectly modify it.

MEMORY ALLOCATION

TYPES OF MEMORY

Operating system allocates the memory block (starting address + block size)


Starting address
Static memory
Heap pointer
Stack pointer
Program code
Global and static variables & objects
Heap
Stack
size known at compilation time
size known at compilation time
all local variables and local objects, including member functions

STATIC MEMORY

- Static memory is sized offline during compilation (before execution).
- All *global variables*, variables outside functions, obtain memory from static memory. All *static variables* will also obtain memory from static memory.
- Global variables act like a local variable but have global scope (no surprise).

```
int a_global = 10;
char another_global = 'A';

void func1() {
    printf("%d\n", a_global);
    a_global_var++;
}

void main() {
    printf("%d\n", a_global);
    a_global_var++;
    printf("%d\n", a_global);
}
```

STATIC MEMORY

- A change made to a static variable will appear within the scope of the function that uses it. The data contained within the variable will be available on later invocations of the function since it is stored statically.
- A static variable only goes out of scope if the program is terminated.

- When might a static local variable be needed?

```
void login() {
    static int counter = 0;      // will be initialized only once
    if (verified())counter++;    // count the # of users logged in
}
```

STATIC VARIABLE VERSUS GLOBAL VARIABLES

- Using a static variable puts the variable declaration in the same place where the variable is actually used. This makes the program easier to read and understand.

- Static variable prevents other functions from accessing the variable if declared locally. As a global variable, all functions can read and modify the variable.

STACK MEMORY

- All non-static "local" variables obtain memory from the stack.
- Stack objects are automatically de-allocated when it passes out of scope.

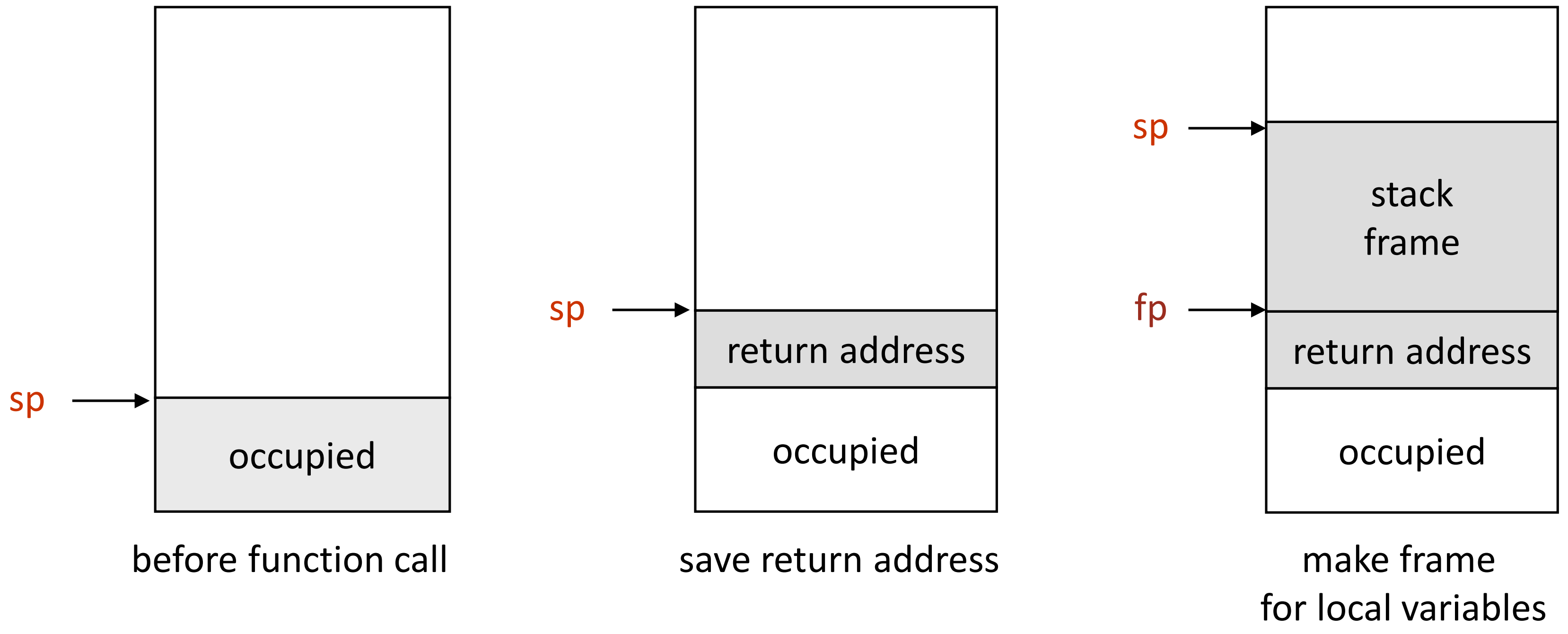
```
int bar(int j) {
    int k = 2;
    j = j + k;
    return j;
}

void main(void) {
    int i = 0;
    i++;
    i = bar(i);
}
```

Stack	
Misc	

STACK MEMORY

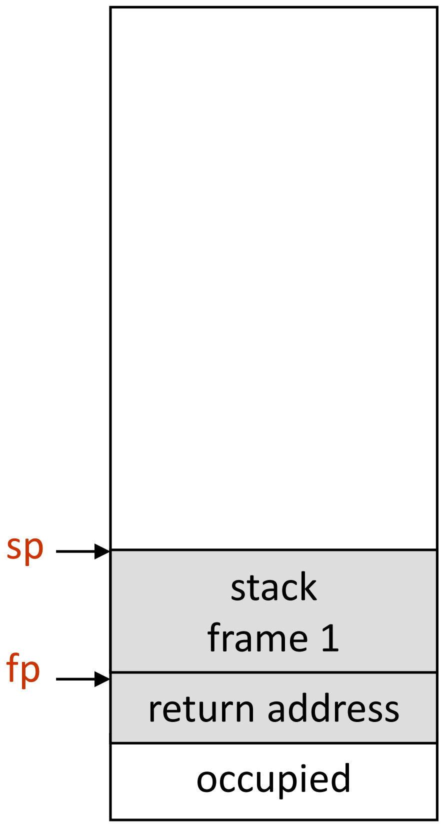
SUPPORTING FUNCTION CALLS AND LOCAL VARIABLES

Two registers are reserved for each function call:
sp: stack pointer
fp: frame pointer, used to access the local variables

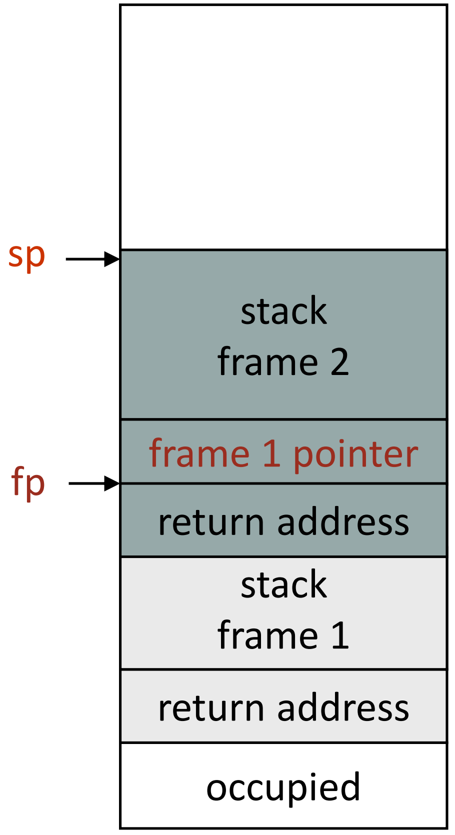


STACK MEMORY

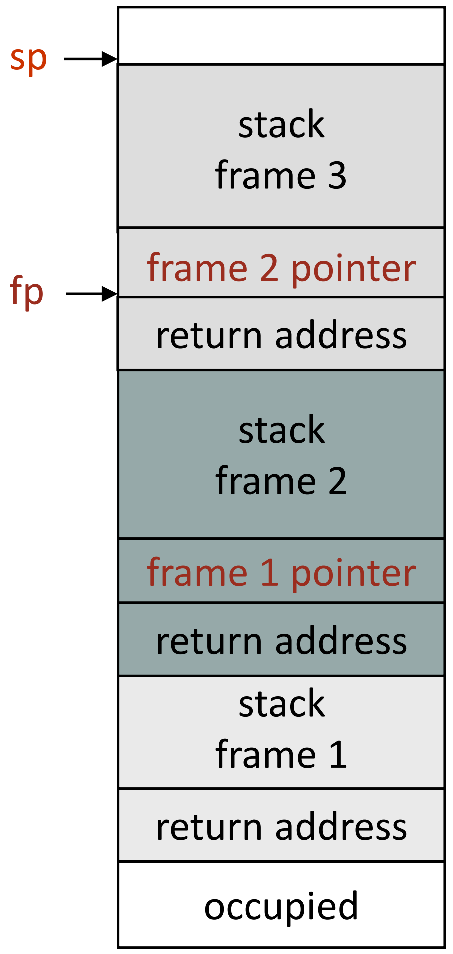
SUPPORTING NESTED AND RECURSIVE FUNCTION CALLS



after first call



after a re-entrance



after another re-entrance

HEAP MEMORY

- Heap memory is used for dynamic (run-time) memory allocation.
- Example:

```
struct contact {                    // define a structure
    char name[30];
    int phone;
} *p;

p = malloc(38);
// ...or...
p = (struct contact *) malloc(sizeof(struct contact));
```

HEAP MEMORY VERSUS STACK MEMORY

When an allocation (say an array) goes out of scope,

- if it comes from the stack, it is automatically popped off the stack, that is, the memory is de-allocated or freed.

- if it comes from the heap, it will not be automatically de-allocated or freed. We have to explicitly use *free* to free the object.

Example:

```
p = (struct contact *) malloc(sizeof(struct contact));

…

free(p);                    // free the object pointed by p to heap
 p = NULL;                  // should try to set p to NULL (why?)
```

HEAP GARBAGE COLLECTION

In Java, automatic garbage collection is used, which is implemented by a **reference counter** in each object. A zero-value implies zero-reference and thus can be garbage collected.

- Pro: Programmers don't need to deal with memory management.
- Con: Automatic garbage collection may not run too often. Then it may come too late or at wrong time. Bad for real time systems.

C programmers must correctly decide when to delete an object. For <u>each</u> **malloc()** operation, one must use a **free** somewhere to delete the memory created by the **malloc()**.

Concerns:

- **Memory leakage**: if a programmer forgets to delete an unused object, the program will eventually run out of memory.
- **Dangling reference**: Try to access an object that has been deleted or go out of scope.

Can you get a dangling reference in Java?