Introduction to C, C++, and Unix/Linux

CS 60

Lecture 11: Memory

Today

→ Memory

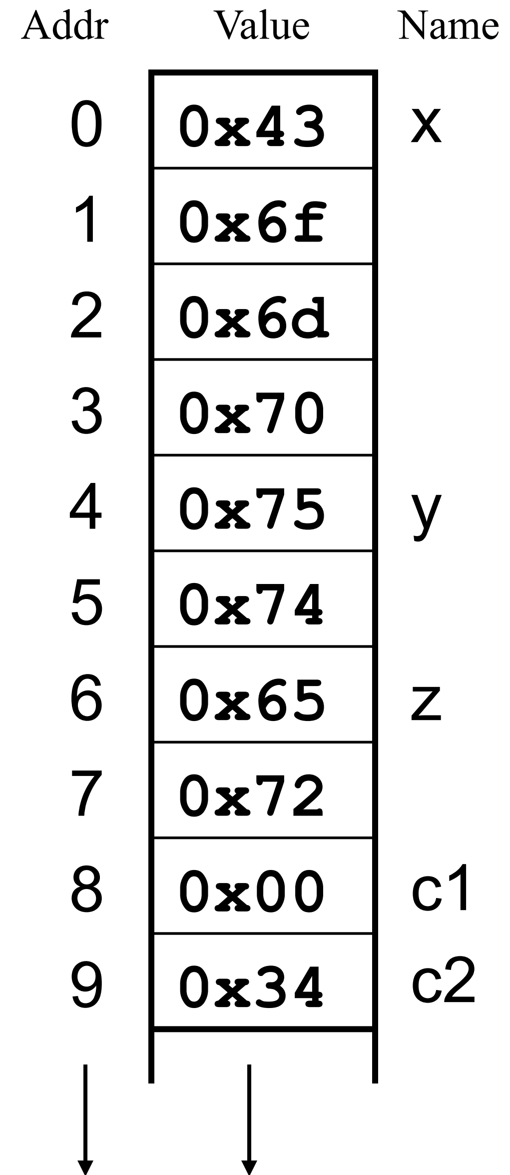
→ Reading [KR] Chapters 1 – 7.

Memory

- Before we talk about pointers, let's be clear on how memory allocation works in C – and how to think about memory in general.
  - Addressable unit of memory
  - Little- or big-endian (order of byte storage)
  - Stack and heap
- Memory can be visualized in several different ways

Memory

- Memory locations have:
  - An address
  - A value (the contents of the memory address)
  - A name that the compiler associates with the memory location



0	1	2	3	4	5	6	7	8	9
43	6f	6d	70	75	74	65	72	00	34
x				y		z		c1	c2

Address	Value	Variable
9	0x34	c2
8	0x00	c1
7	0x72	
6	0x65	z
5	0x74	
4	0x75	y
3	0x70	
2	0x6d	
1	0x6f	
0	0x43	x

What are the values of x and y?

```
char x, y;    x = 0x43
              y = 0x75

int x, y;     x = 0x706d6f43
              y = 0x72657475

float x, y;   x = ...
              y = ...
```

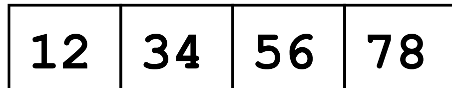
What order are bytes stored?

- **Big-endian** – the most significant byte has the lowest address ("big end first")
- **Little-endian** – The least significant byte has the lowest address ("little end first")
- Historically, most mainframes have been big-endian, and PCs are little-endian. This *does* affect portability!
  - CSIL machines (all Intel-based computers) are little-endian
  - Motorola processors (Macs) are big-endian

```
int x = 0x12345678;
```

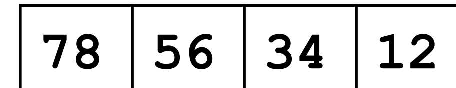
MSB LSB

Big-endian



lower address → higher address

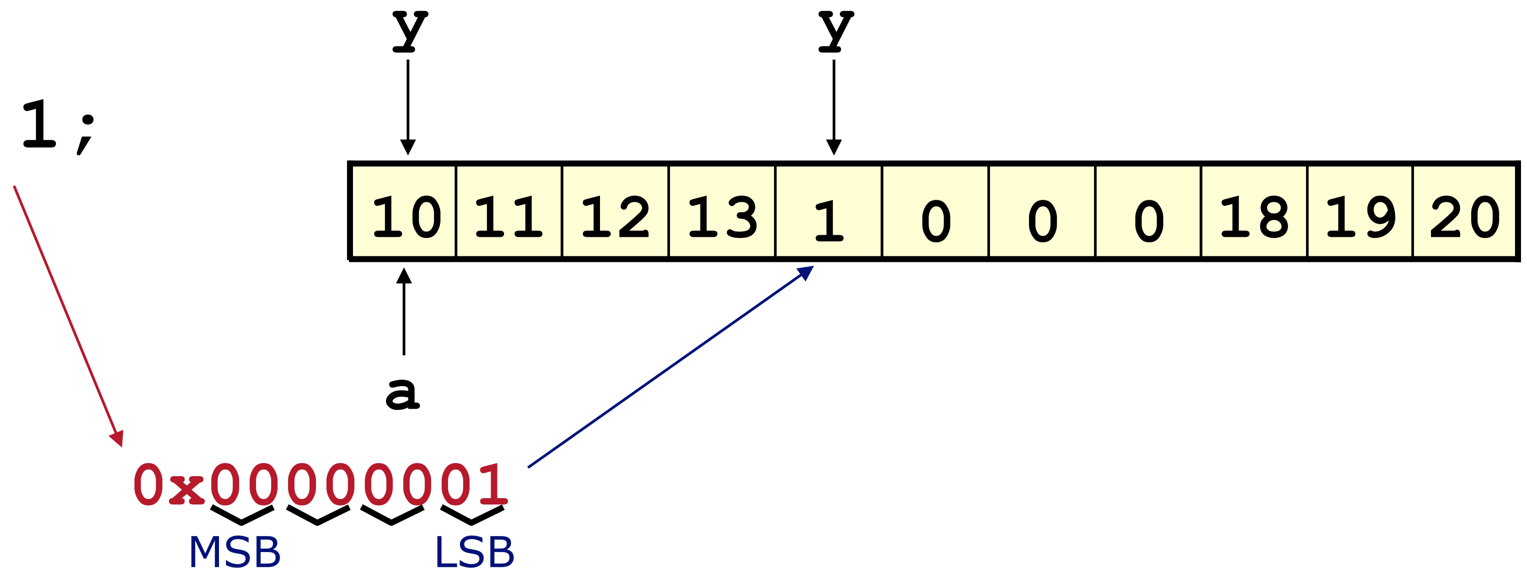
Little-endian



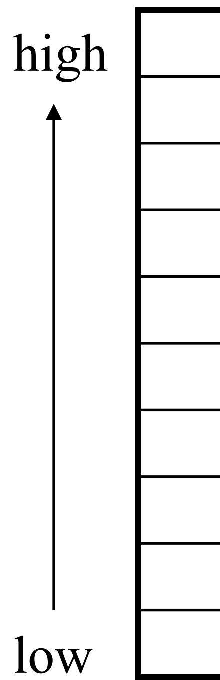
lower address → higher address

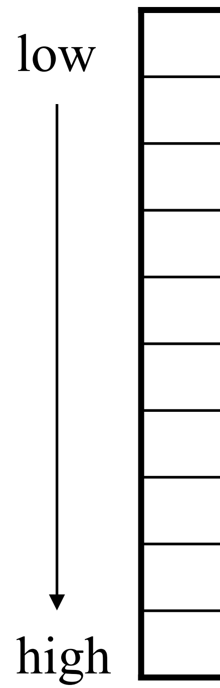
Example (using pointers)

```
unsigned char a[] =
   {10,11,12,13,14,15,16,17,18,19,20};
int *y = a;
y++;
*y = 1;
```

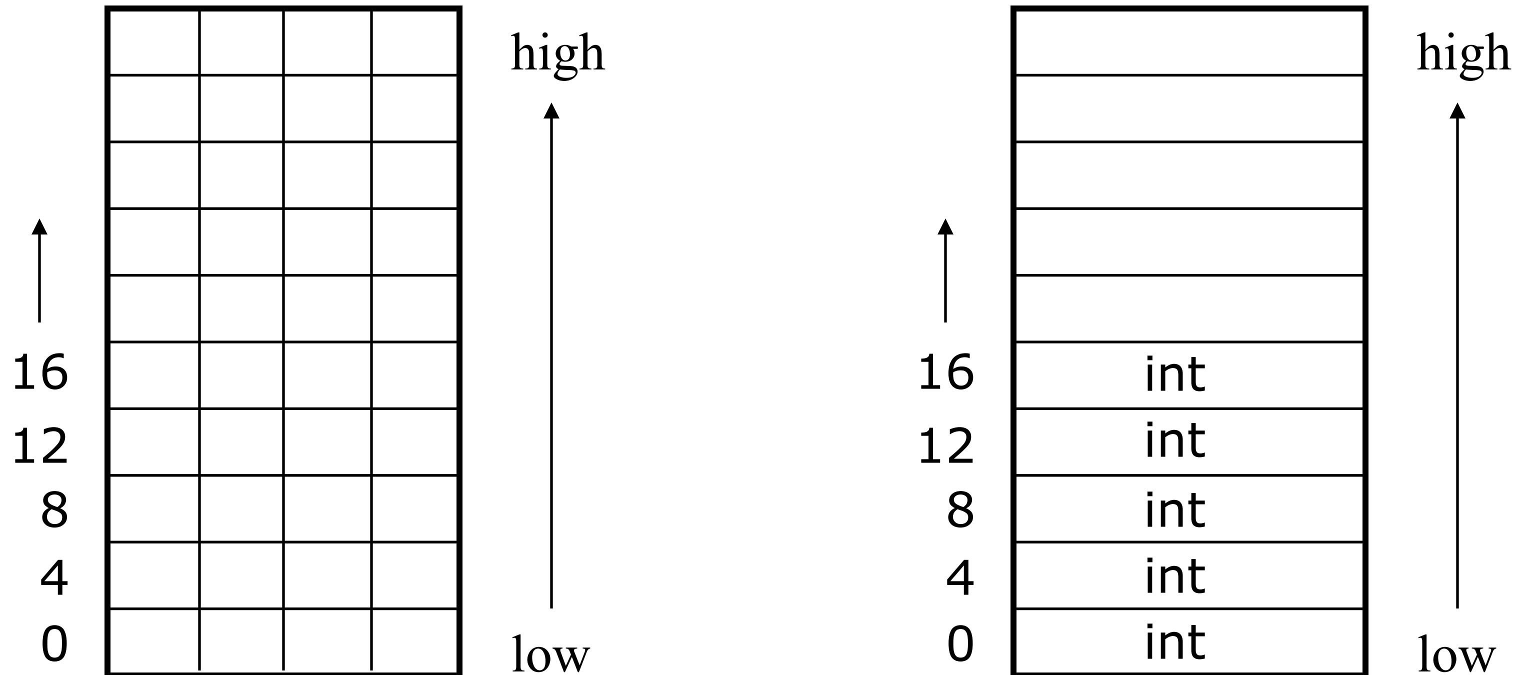


Visualizing memory: direction



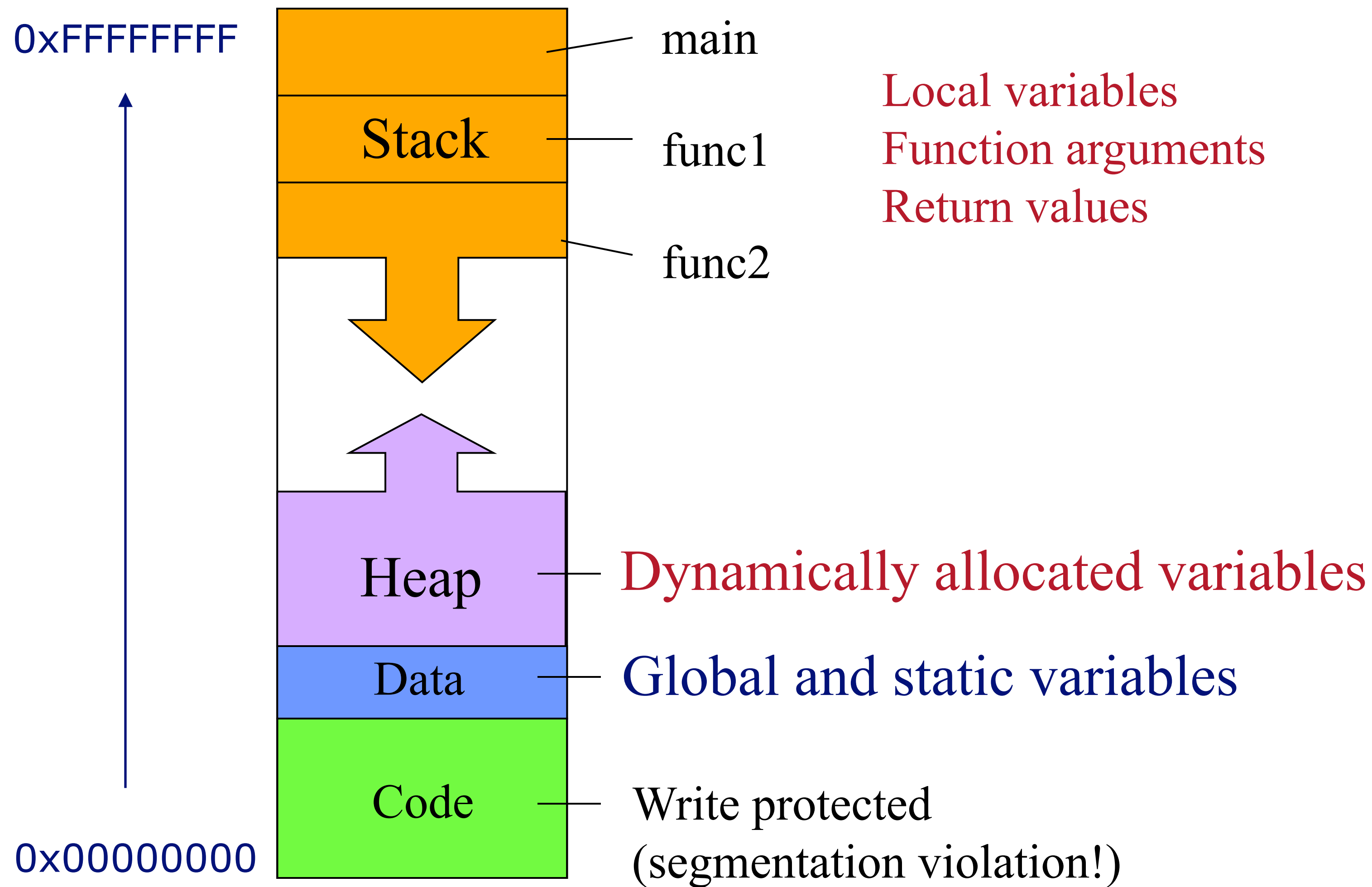


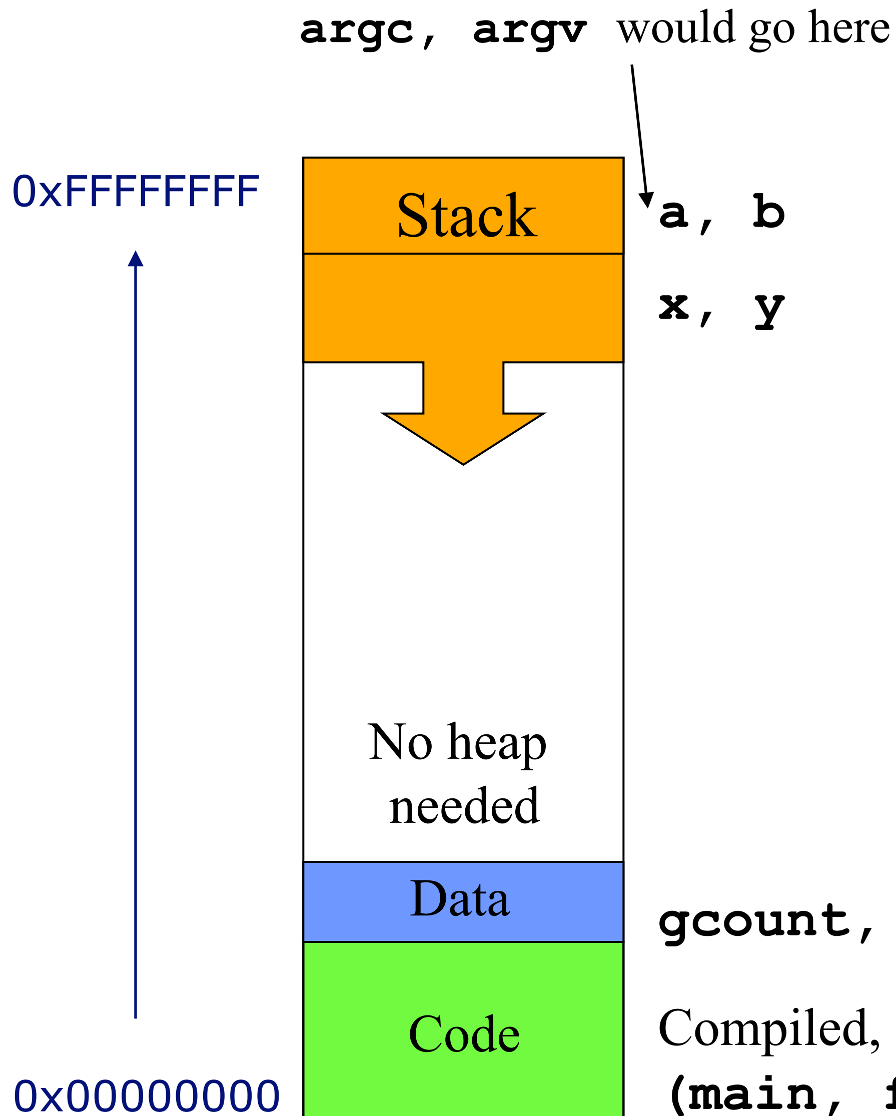
Visualizing memory: grouping



Bottom line: People describe memory in many different ways. You have to learn to think clearly about what is being stored: where, how, and why

Address space of a C program





Compiled, executable binary program
(main, func)

```
int gcount=0;

int func(int x)
{
  int y = x*x;
  return(y);
}

int main(void)
{
  int a=2, b;
  static int c=0;
  b = func(a+c);
  ....
}
```

Function return values

- Remember, C is call by value
  - This is both for passing variables into functions and for getting return values from functions

- Example...

```
int func(int x)
{
   ...
   return(y);
}

int main()
{
   ...
   b = func(a+c);
   ....
}
```

2. **func** uses that stack memory as its local variable **x**

3. **y** is evaluated, then copied to an integer-sized location on the stack

4. That stack value is copied to b (also a stack integer)

1. **a+c** is evaluated, then copied to an integer-sized location on the stack

```
struct data {
  int x; int y; int *data;
};

struct data func(struct data in)
{
  in.x = 0;
  in.y = 0;
  in.data = (int *)malloc(100);

  return(in);
}
```

How many bytes will **`func`** reserve on the stack for its return value?

How many bytes is the variable **`in`**?

```
struct data {
  int x; int y; int *data;
};

int main(void)
{
  struct data test;
  test.x = test.y = 100;
  test.data = -1;

  func(test);
  ...
}
```

← What are the values of `test` after this call?

`100, 100, -1`