C Review

Notes from Dr. Charlie Obimbo and Dr. Andrew Hamilton-Wright
Revised by Yan Yan.

Contents

1. Storage classes and scopes
2. Tips for C programming
3. Review of pointers

Declaration:

- int v = 0; /* declared at some place in your program */
- What can you see from the declaration?

storage class : auto, static, extern, register

storage class	Storage Place	Scope	Lifetime
auto	RAM	Local	Within function
extern	RAM	Global	Till the end of the main program. Maybe declared anywhere in the program
static	RAM	Local	Till the end of the main program. Retains value between multiple functions call
register	Register	Local	Within function

Storage class:

- Auto: declared inside a block, exists only when the block is entered, and disappears when execution leaves the block.

```
{
   int x, y; …


}
```

x and y alive
Accessable in
this block

- Static: accessible in the block where it is declared, exists and retains its value in whole program cycle.

```
void f(){
        int x = 0;
        printf(“%d\n”, x++);
}

int main(){
        f();
        f();
        f();
}
```

```
void g(){
        static int x = 0;
        printf(“%d\n”, x++);
}

int main(){
        g();
        g();
        g();
}
```

- What output you will get from the two programs?

- Extern: accessible and exists in whole program file and program cycle. Define a global variable exactly once and use external declarations everywhere else.

```
#include <stdio.h>
// declaring a variable with extern variable

extern int var;

int main(){
        // defining the extern variable
        int var=10;
        printf("%d",var);
}
```

- Register: frequently used variables for efficiency purpose.
- It cannot declare global register variables
- The compiler may ignore register declaration.
- Cannot take the address of a register variable (No pointers to point to the address of it).

Declaration:

- int   v = 0; /* declared at some place in your program */
- What can you see from the declaration?

type:	value domain
value:	current value ∈ value domain

```
int x = 9;
x = x/9;
```

What is the value of x?

Declaration:

- int v = 0; /* declared at some place in your program */
- What can you see from the declaration?

name:	symbolic identifier
location:	memory address
size:	how many bytes it occupies

Tips for C Programming:

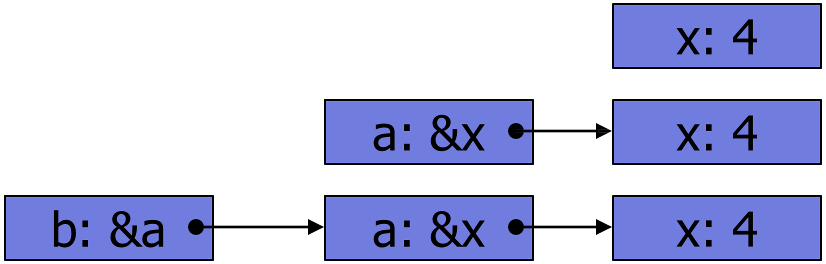
- Do not change a loop variable inside a for loop block.
- All flow control primitives (if, else, while, for, do, switch, and case) should be followed by a block, even if it is empty.
- Use break and continue instead of goto.
- Do not have overly complex functions.
- Indent to show program structure (better readability).
- Parenthesize to resolve ambiguity.

Pointers: what for?

- A pointer is defined by its type and holds a value
- value: indicates where in memory the pointer refers to
  - always a memory address
- type: indicates what in memory the pointer refers to
  - almost always indicates the object’s size – there are two exceptions:
    - Pointer to void
    - Pointer to function

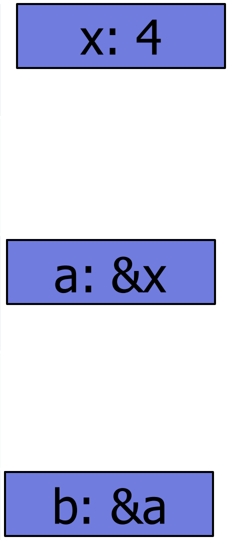
Pointers Exmaple

```
int x = 4;
int *a = &x;
int **b = &a;
```



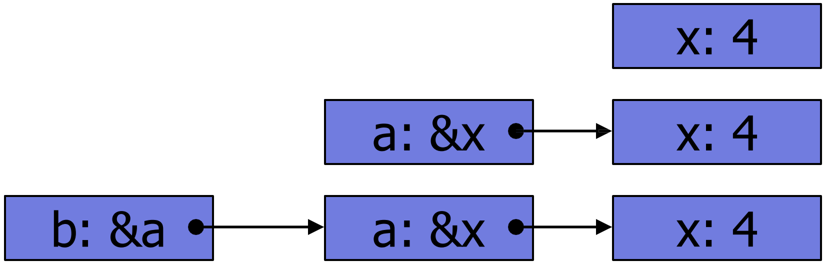
Memory

Address	Value
000100	4
…	
000104	000100
…	
000120	000104



Pointers Exercise:

```
int x = 4;
int *a = &x;
int **b = &a;
```



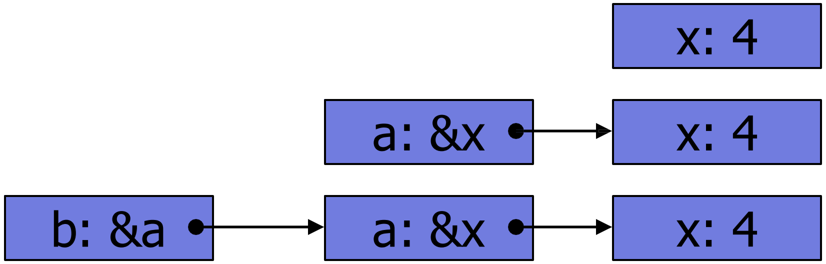
Let’s fill the table

x	4	a	addr(x)	b	addr(a)
&x	addr(x)	&a	addr(a)	&b	addr(b)
*x		*a		*b	
*(&x)		**a		**b	
				***b	

Note some may be illegal

Pointers Exercise:

```
int x = 4;
int *a = &x;
int **b = &a;
```



x: 4

a: &x → x: 4

b: &a → a: &x → x: 4

x	4	a	addr(x)	b	addr(a)
&x	addr(x)	&a	addr(a)	&b	addr(b)
*x	illegal	*a	4	*b	addr(x)
*(&x)	4	**a	illegal	**b	4
				***b	illegal

*b == a == &x
**b == *a == x

The unary operators * and &

```
int x, y, *px;

 x = 10;
 px = &x;
 y = *px;
```

- The unary operator & gives the address of an object.
- The unary operator * treats its operand as the address of a memory cell, and accesses the cell to get the contents.

The unary operators * and &

```
int  x,  y;
 int *px;
 x = 10;
 px = &x;
 *px += 1;
 (*px)++;
```

- Declaration int *px; means that *px is an int, or px is a memory cell containing a pointer to a variable of int.
- *px can be on the left side of an assignment.
- In (*px)++, the parentheses are required.

Function Arguments and Pointers

```
int main (void) {

…
            int m,n,x;
            m=2;
            n=1;
…
            x=add(m,n);
…

}
```

```
int add(int a, int b){
            return(a+b);
}
```

m	n	x	

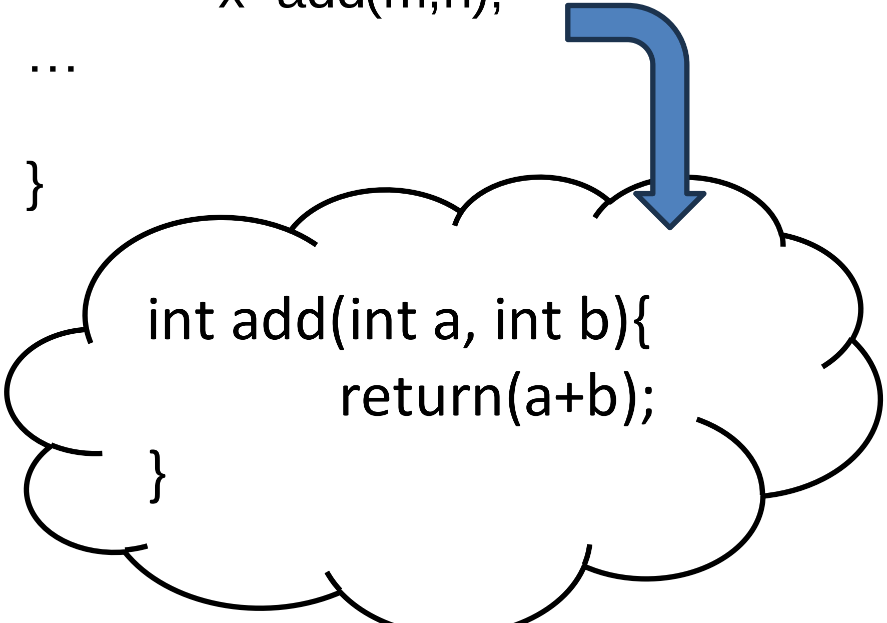
Function Arguments and Pointers

```
int main (void) {

…
        int m,n,x;
        m=2;
        n=1;
…
        x=add(m,n);
…

}
```

2	1	x	



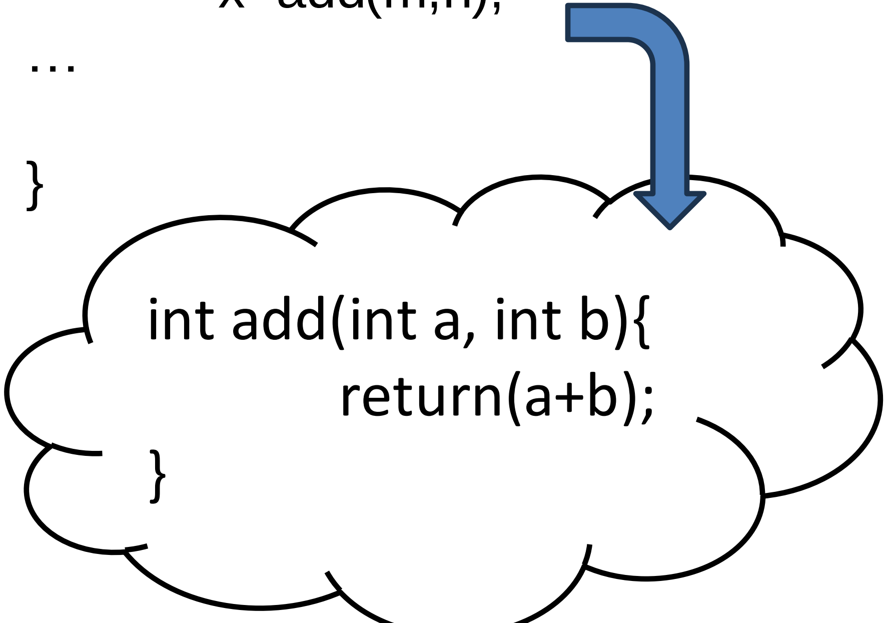
```
int add(int a, int b){
        return(a+b);
}
```

Function Arguments and Pointers

```
int main (void) {

…
            int m,n,x;
            m=2;
            n=1;
…
            x=add(m,n);
…

}
```



```
int add(int a, int b){
            return(a+b);
}
```

2	1	3	
	2	1	

Function Arguments and Pointers

```
int main (void) {

…
          int m,n,x;
          m=2;
          n=1;
…
          x=add(m,n);
…

}
```

```
int add(int a, int b){
          return(a+b);
}
```

2	1	3	

call by value

Function Arguments and Pointers – Call by Reference

```
int main (void) {

…
          int m,n,x;
          m=2;
          n=1;
…
          change_value(&m,&n);
…

}
```

```
int change_value(int *a, int *b){
          *a=20;
          *b=10;
}
```

m	n	x	

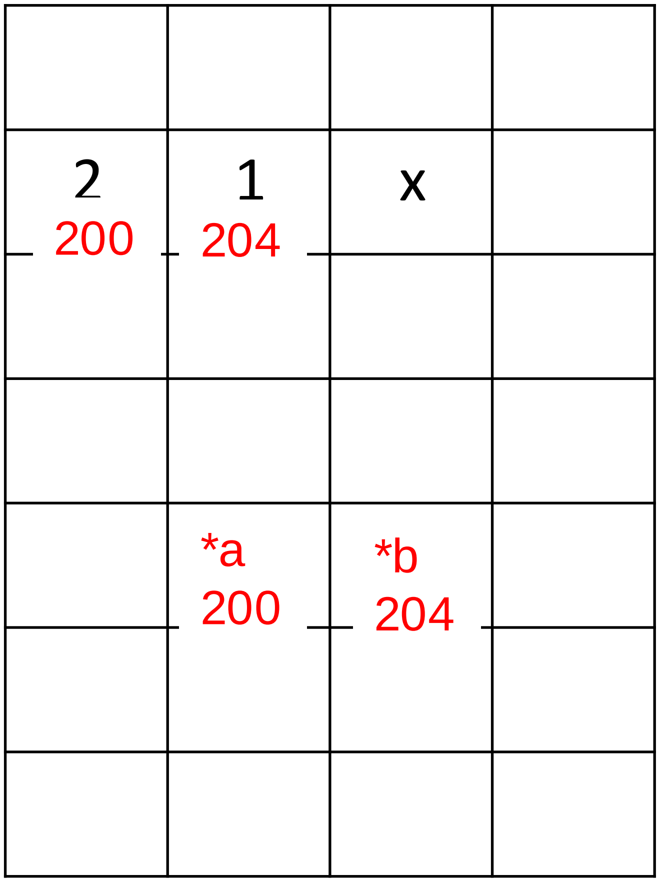
Function Arguments and Pointers – Call by Reference

```
int main (void) {

…
            int m,n,x;
            m=2;
            n=1;
…
            change_value(&m,&n);
…

}
```

```
int change_value(int *a, int *b){
            *a=20;
            *b=10;
}
```



2 200 | 1 204 | x

*a 200 | *b 204

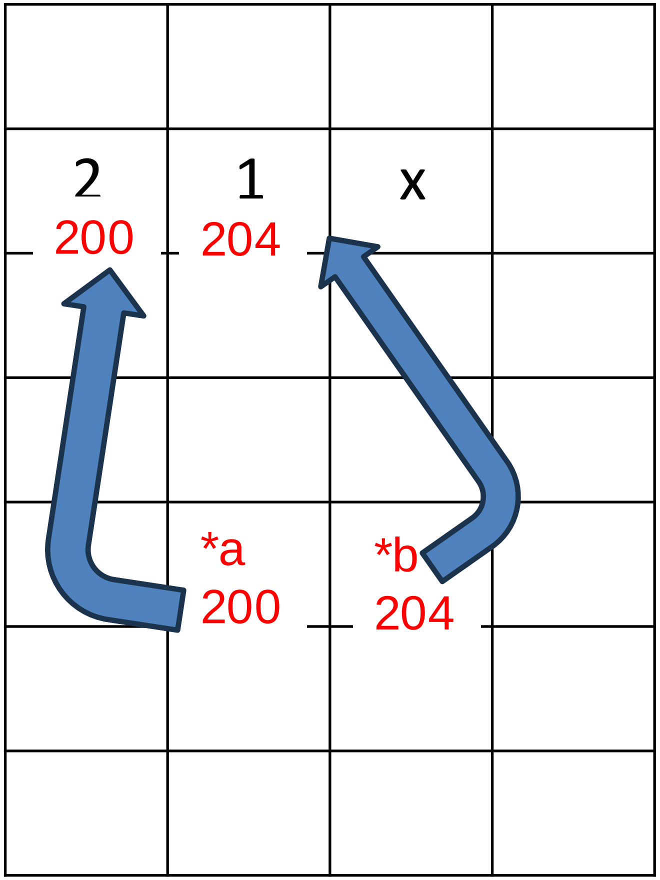
Function Arguments and Pointers – Call by Reference

```
int main (void) {

…
            int m,n,x;
            m=2;
            n=1;
…
            change_value(&m,&n);
…

}
```

```
int change_value(int *a, int *b){
            *a=20;
            *b=10;
}
```



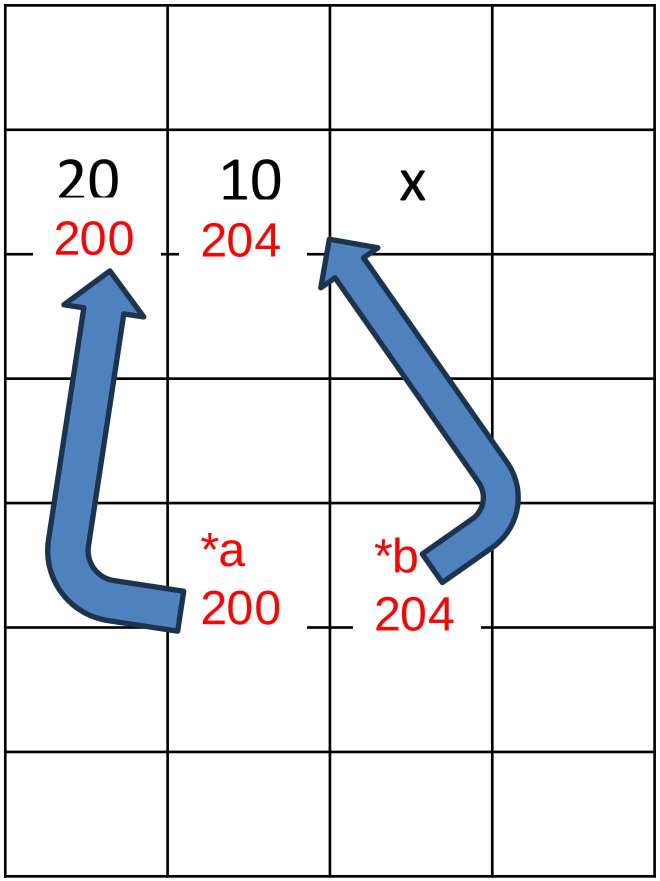
Function Arguments and Pointers – Call by Reference

```
int main (void) {

…
            int m,n,x;
            m=2;
            n=1;
…
            change_value(&m,&n);
…

}
```

```
int change_value(int *a, int *b){
            *a=20;
            *b=10;
}
```



20 10 x

200 204

*a 200

*b 204

Function Arguments and Pointers – Call by Reference

```
int main (void) {

…
            int m,n,x;
            m=2;
            n=1;
…
            change_value(&m,&n);
…

}
```

```
int change_value(int *a, int *b){
            *a=20;
            *b=10;
}
```

20 200	10 204	x	

Pointers and Arrays

```
int x[5] = {1, 3, 5, 7, 9};
 int *px;
 int y, z;
 px = &x[0];            //Set px to point to x[0].
 y = *px;               //Assign the content of x[0] to y.

 px = x;                //Set px to point to x[0], which is the
                        //beginning of x.
z = *(px+1);            // same as z = x[1].
```

- When p is a pointer to an array, p+1 points to the second element, p+2 points to the third element, …

	Value	address
x[0]	1	200
x[1]	3	204
x[2]	5	208
x[3]	7	212
x[4]	9	216
	*px 200	
	y 1	
	z 3	

Pointers and Arrays

- Array name is a pointer as well.
- In an array X, element index i
  - Address: &X[i] or (X+i)
  - Value: X[i] or *(X+i)
- First element address is the base address of the array
  - E.g. X or &X[0]
- Increment of Array name is illegal
  - E.g. X++ // illegal

```
int *pX = X;
pX++; // legal
```

Pointers and Arrays – Exercise

- In an array, int numbers[5], what does numbers represent?
  - A. numbers[4]
  - B. numbers[0]
  - C. &numbers[4]
  - [✓ D]. &numbers[0]
  - E. Illegal