

Pointers


(review + memory allocation)

Yung-Hsiang Lu

Pointers in C

- Understanding pointers is **critical** in learning C.
- In a computer, data are stored in memory as address - data pairs
- A pointer's data field is an address.
- Programmers have **no** control of the addresses of data.
- Programmers **can** control the data of pointers, i.e. where they point to.

Address	Data
0X00000000	...
...	...
0x08001F00	'a'
...	...
0X1A0088F0	642
...	...
0X1BFF0000	0X1A0088F0



Pointer Operation (review)

```
int * p;          /* p is a pointer to an integer */
```

```
int x = 123;
```

```
int y = 987;
```

```
p = & x;          /* p points to x */
```

p → x

```
*p = 456;         /* x is 456 now */
```

```
p = & y;          /* p points to y */
```

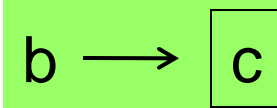
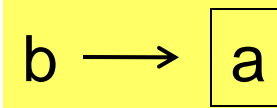
p → y

```
*p = -765;        /* y is -765 now */
```

Address	Data
somewhere (&x)	456
...	...
somewhere (&p)	& x
somewhere (&y)	987

Address	Data
somewhere (&x)	456
...	...
somewhere (&p)	& y
somewhere (&y)	-765

```
Terminal
File Edit View Terminal Help
[Ubuntu Linux ] more pointer1.c
#include <stdio.h>
int main(int argc, char * argv[])
{
    int a = 100;
    int * b;    /* b is a pointer to an integer */
    b = &a;     /* b points to a */
    *b = 200;   /* The value where b points to
                  is changed to 200 */
    printf("a's value is %d\n", a);
    int c = 300;
    b = &c;     /* b points to c */
    * b = 400; /* The value where b points to
                  is changed to 400 */
    printf("c's value is %d\n", c);
    return 0;
}
[Ubuntu Linux ] gcc pointer1.c -o pointer1
[Ubuntu Linux ] ./pointer1
a's value is 200
c's value is 400
[Ubuntu Linux ]
```



Common Mistake

A program cannot control addresses.

```
65 = 123;
```


```
/* cannot assign value 123 to the address 65 */
```

```
int * p = 86;
```

```
/* warning message, the
```

```
address is likely occupied by another program. */
```

Address	Data
86	...
somewhere (&p)	86




```
double d = 0.84;
```

```
int * p = & d;      /* warning message */
```

Modify the Argument

void means return nothing



```
#include <stdio.h>
void incr(int a)
{
    printf("input= %d\n", a);
    a ++;
    printf("before function ends = %d\n", a);
}

int main(int argc, char * argv[])
{
    int x = 8;
    printf("before calling incr, x = %d\n", x);
    incr(x);
    printf("after calling incr, x = %d\n", x);
    return 0;
}
```

Console

<terminated> Function [C/C++ Local Ap]

before calling incr, x = 8
input= 8
before function ends = 9
after calling incr, x = 8

**x is still 8 after
calling the function**

error: inv...expression

Writable

Smart Insert

15 : 47

Call by Value

A function **copies** the **value** to the function's argument.

```
void incr(int a) ...
```

```
int x = 8;
```

```
incr(x);      /* copy x's value to a */
```

```
a ++;        /* a becomes 9, x is still 8 */
```

similar to

```
int x = 8;
```

```
int y = x;
```

```
y ++;        /* y is 9, x is still 8 */
```

Address	Data
somewhere (&x)	8
somewhere (&a)	8 → 9

Address as Argument

The screenshot shows the Eclipse IDE with a C++ project. The main editor displays the source code for `function4.c`. The code defines a function `incr` that takes a pointer to an integer and increments the value it points to. The `main` function initializes a variable `x` to 8, prints its value, calls `incr` with the address of `x`, prints the value again, and returns 0. Annotations include a red arrow pointing to the asterisk in the function signature and another pointing to the ampersand in the function call. The console window on the right shows the output of the program, confirming that the value of `x` is updated to 9 after the function call. A red arrow points to the variable `x` in the `main` function.

```
#include <stdio.h>
void incr(int * a)
{
    printf("input= %d\n", * a);
    (*a) ++;
    printf("before function ends = %d\n", * a);
}

int main(int argc, char * argv[])
{
    int x = 8;
    printf("before calling incr, x = %d\n", x);
    incr(& x);
    printf("after calling incr, x = %d\n", x);
    return 0;
}
```

Console Output:

```
<terminated> Function [C/C++ Local Appli]
before calling incr, x = 8
input= 8
before function ends = 9
after calling incr, x = 9
```


Call by Address

A function **copies** the **address** to the argument.

```
void incr(int * a) ...
```

```
int x = 8;
```

```
incr(& x);    /* copy x's value to a */
```

```
(*a) ++;      /* a becomes 9, so is x */
```


similar to

```
int x = 8;
```

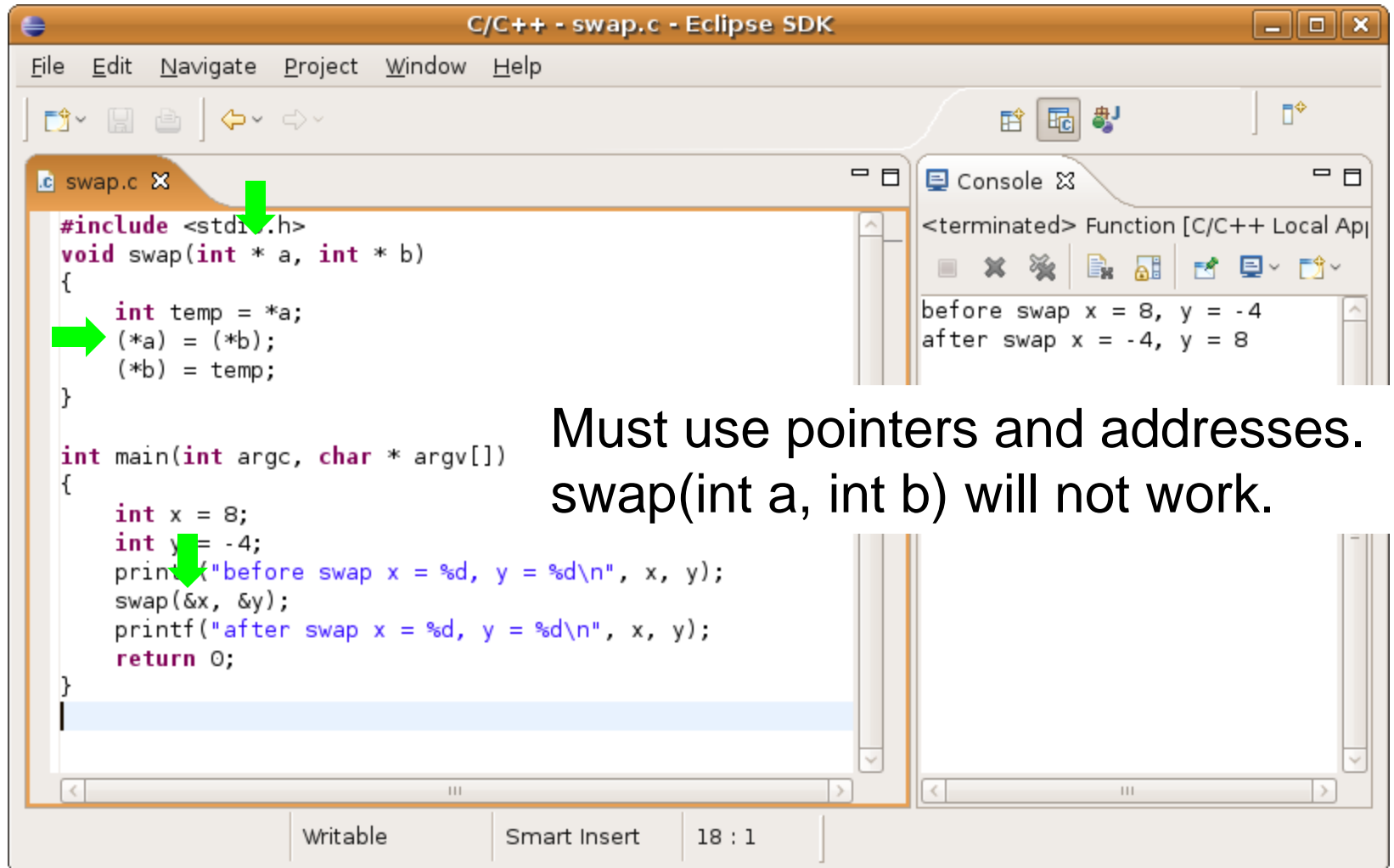
```
int * y = & x;
```

```
(*y) ++;      /* x is 9 */
```

Address	Data
somewhere (&x)	8
somewhere (&a)	& x



Swap Function



The screenshot shows the Eclipse IDE with a C program named `swap.c`. The code defines a `swap` function that takes two integer pointers and swaps their values. The `main` function initializes `x` to 8 and `y` to -4, prints their values, calls `swap(&x, &y)`, and prints the values again. The console output shows the values before and after the swap.

```
#include <stdio.h>
void swap(int * a, int * b)
{
    int temp = *a;
    (*a) = (*b);
    (*b) = temp;
}

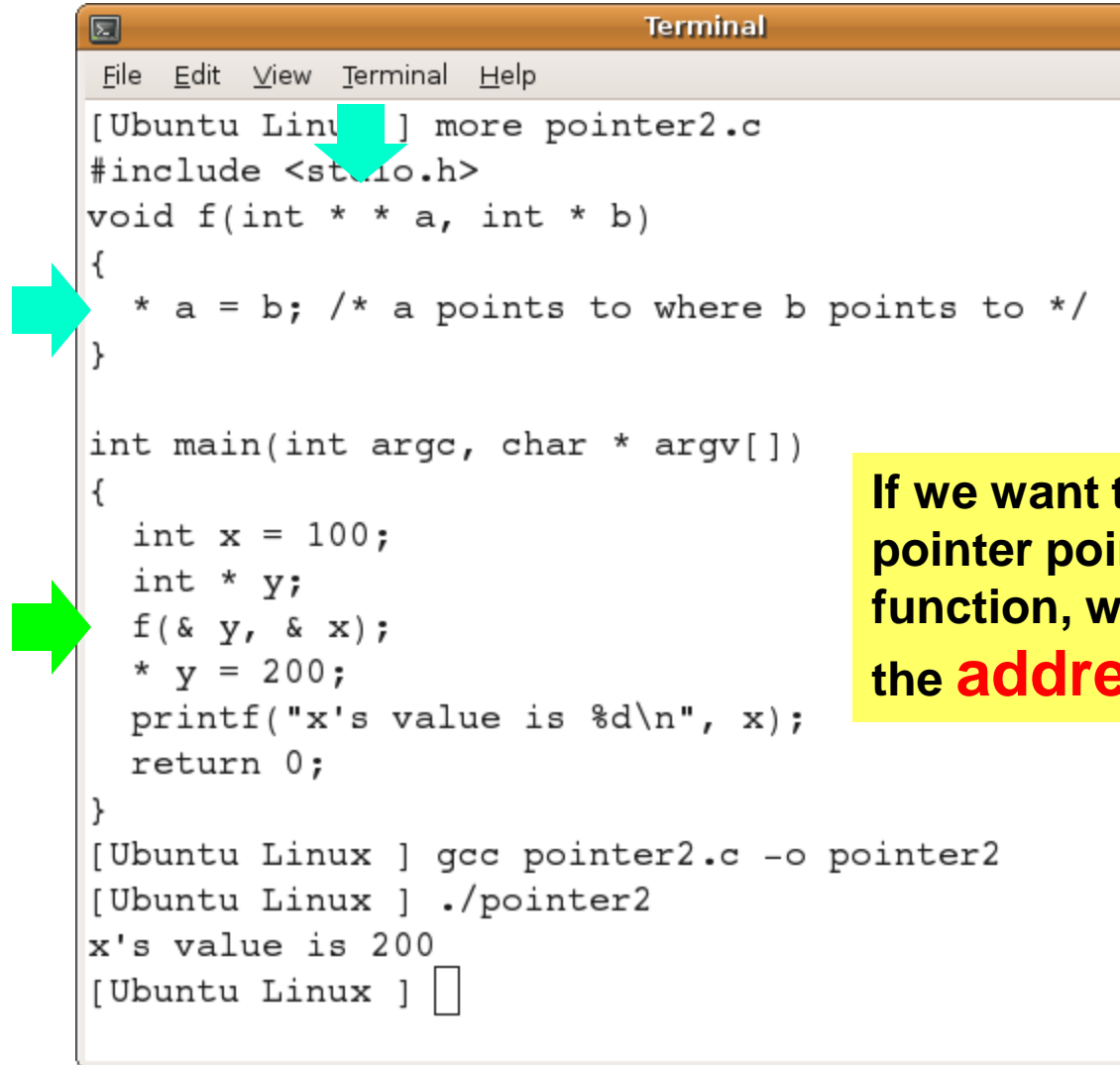
int main(int argc, char * argv[])
{
    int x = 8;
    int y = -4;
    printf("before swap x = %d, y = %d\n", x, y);
    swap(&x, &y);
    printf("after swap x = %d, y = %d\n", x, y);
    return 0;
}
```

Console Output:

```
<terminated> Function [C/C++ Local Appl
before swap x = 8, y = -4
after swap x = -4, y = 8
```

Must use pointers and addresses.
`swap(int a, int b)` will not work.

Pointer as Function Argument



```
[Ubuntu Linux] more pointer2.c
#include <stdio.h>
void f(int ** a, int * b)
{
    * a = b; /* a points to where b points to */
}

int main(int argc, char * argv[])
{
    int x = 100;
    int * y;
    f(& y, & x);
    * y = 200;
    printf("x's value is %d\n", x);
    return 0;
}

[Ubuntu Linux] gcc pointer2.c -o pointer2
[Ubuntu Linux] ./pointer2
x's value is 200
[Ubuntu Linux]
```

Address	Data
&x	100
&b	& x
&a	& y
&y	change

If we want to change where a pointer points to inside a function, we have to provide the **address** of the pointer.

General Rules

To change the value inside a function and to keep this change after the function returns

- The caller must use the **address**

`f(&x);`

- The function's argument adds **one ***

`f(int *x) { ... }`

- If the argument is already a pointer, use two *

`f(int **x) { ... }`

- Inside the function, changes the value by adding *

`* a = b;`

Allocate Memory in a Function

```
#include <stdio.h>
#include <stdlib.h>
const int arraySize = 10;
void printArray(int * a)
{
    int index;
    for (index = 0; index < arraySize; index ++)
        { printf("%d ", a[index]); }
    printf("\n");
}
void f(int * * a)
{
    * a = malloc(arraySize * sizeof(int));
}
int main(int argc, char * argv[])
{
    int * x;
    f(& x);
    int index;
    for (index = 0; index < arraySize; index ++)
        { x[index] = index * 10; }
    printArray(x);
    free (x);
    return 0;
}
[Ubuntu Linux ]
```