Pointers

Instructor: Tsung-Che Chiang
Department of Computer Science and Information Engineering
National Taiwan Normal University

tcchiang@ieee.org
https://moodle3.ntnu.edu.tw/course/view.php?id=42615

Warm-up Exercise

- Write a function to swap the values of two variables.

```
void swap(                  )
{

}
// ------------------------------------
int main()
{
    int a = 10, b = 20;
    printf("a = %d, b = %d\n", a, b); // a = 10, b = 20
    swap(                );
    printf("a = %d, b = %d\n", a, b); // a = 20, b = 10
}
```

Warm-up Exercise

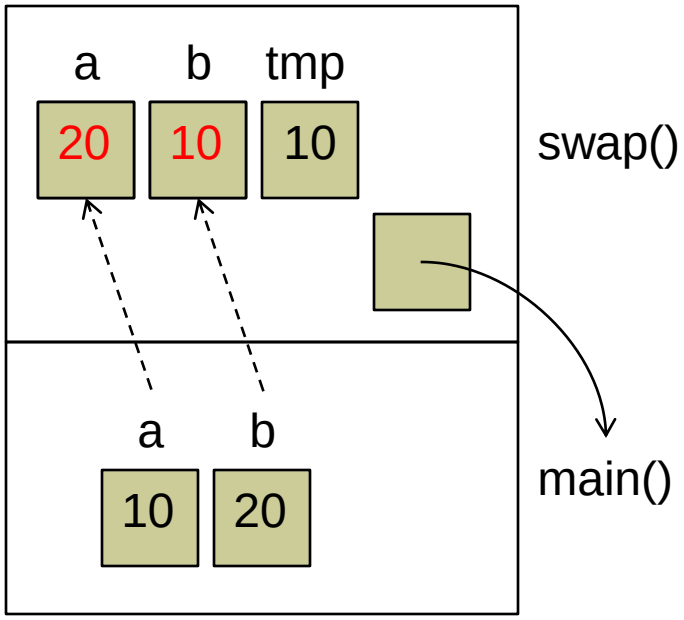
- You may try this:

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
// ------------------------------------
int main()
{
    int a = 10, b = 20;
    printf("a = %d, b = %d\n", a, b); // a = 10, b = 20
    swap(a, b);
    printf("a = %d, b = %d\n", a, b); // a = 10, b = 20
}
```

Warm-up Exercise

- Why doesn't it work?

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
// --------------------------------
int main()
{
    int a = 10, b = 20;
    printf("a = %d, b = %d\n", a, b);

    swap(a, b);

    // a = 10, b = 20
    printf("a = %d, b = %d\n", a, b);
}
```



Warm-up Exercise

- Currently we only learn to pass parameters by value.
  - Only the value is passed, and it is stored in a local copy in the called function.
  - We have another mechanism, pass-by-reference, in C++. We may talk about it later in the next semester.

- We need a method to access the variables outside the function.

Pointers: “I’m coming~~~”

Definition

- An `int` variable stores an integral value, a `float`/`double` variable stores a real value, and a pointer variable stores a memory address.
  - Recall what the “memory address” is in the course “計算機概論”.

Definition

- To define a pointer, use * in the definition.

```
int main()
{
    int age = 0;
    float average = 0;

    int *p; // a pointer storing the address of an int variable
    float *q; // a pointer storing the address of a float variable
    int *r, s; // r is a pointer, but s is an int variable.
    int *x, *y, *z; // all pointers
}
```

Address Operator &

- Okay, now I know that a pointer stores an address of a variable.
- How can we get the address of a variable?

```
int main()
{
    int age = 32;
    float average = 1.234;
    int *p; // a pointer storing the address of an int variable
    float *q; // a pointer storing the address of a float variable

    p = &age;
    q = &average;

    p = &average; // error: incompatible type int * vs. float *
    q = &age; // error: incompatible type float * vs. int *
}
```

Address Operator &

- Can I know the value of a pointer?

```
int main()
{
    int age = 32;
    float average = 1.234;
    int *p; // a pointer storing the address of an int variable
    float *q; // a pointer storing the address of a float variable

    p = &age;
    q = &average;

    printf("Age: %d, address: %p, p: %p\n", age, &age, p);
    printf("Average: %f, address: %p, q: %p\n", average, &average, q);
}
```

Age: 32, address: 000000000061FE0C, p: 000000000061FE0C
Average: 1.234000, address: 000000000061FE08, q: 000000000061FE08

Note. Actually `%p` expects pointers of `void *`, but here we just let it go.

```
printf("%p", static_cast<void *>(&age));
```

Address Operator &

- A pointer is a variable and also occupies memory space.
- We can also get the address of a pointer.

```c
int main()
{
    int age = 32;
    float average = 1.234;
    int *p; // a pointer storing the address of an int variable
    float *q; // a pointer storing the address of a float variable

    p = &age;
    q = &average;

    printf("Address of age: %p\n"
           "Address of average: %p\n"
           "Address of p: %p\n"
           "Address of q: %p\n", &age, &average, &p, &q);
}
```

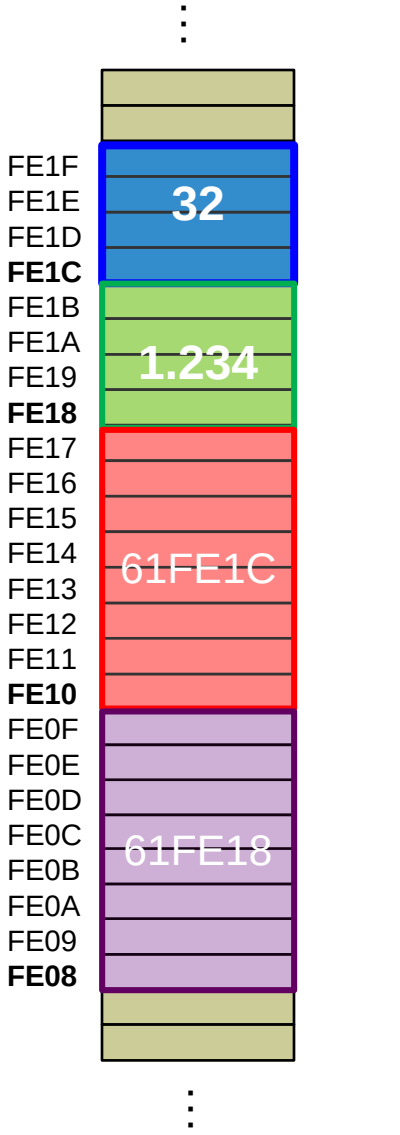
```
Address of age: 000000000061FE1C
Address of average: 000000000061FE18
Address of p: 000000000061FE10
Address of q: 000000000061FE08
```

Address Operator &

1. This is just an example. The memory addresses can be different in your case.
2. The sizes of data types may also be different.
3. We do not guarantee that these variables occupy contiguous memory space.

```c
int main()
{
    int age = 32;
    float average = 1.234;
    // a pointer storing the address of an int variable
    int *p;
    // a pointer storing the address of an float variable
    float *q;

    p = &age;
    q = &average;

    printf("Address of age: %p\n"
           "Address of average: %p\n"
           "Address of p: %p\n"
           "Address of q: %p\n", &age, &average, &p, &q);
}
```

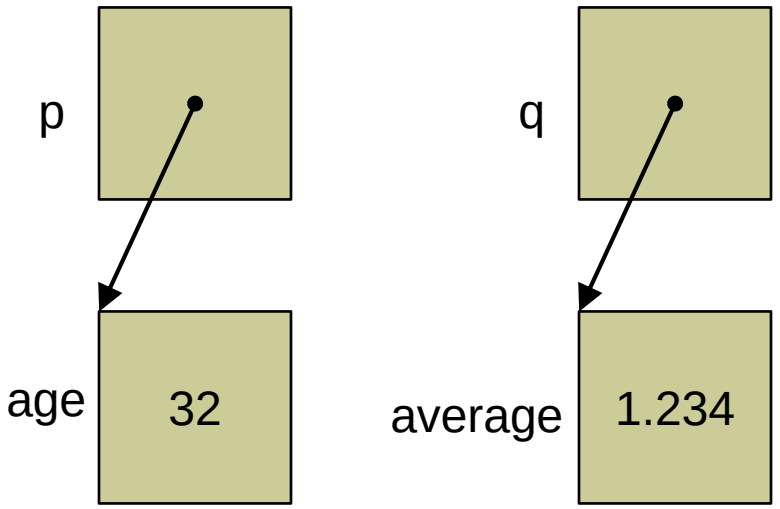
Address of age: 000000000061FE1C
Address of average: 000000000061FE18
Address of p: 000000000061FE10
Address of q: 000000000061FE08



Address Operator &

- We do not often use the actual addresses to indicate the relationship between the pointers and the pointees.
- A common illustration is to use an arrowed line from the pointer to the pointee.

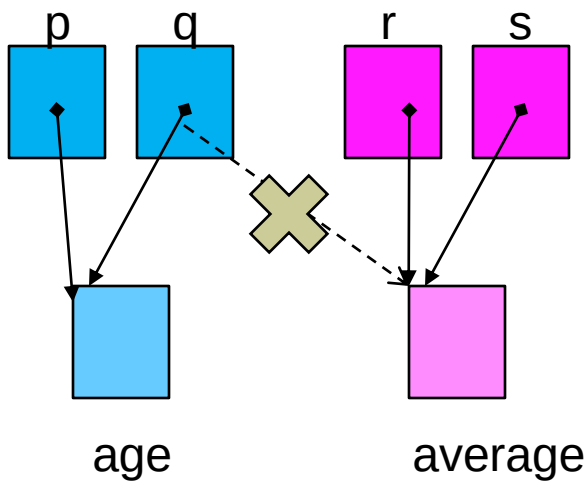
```
int main()
{
   int age = 32;
   float average = 1.234;
   int *p;
   float *q;

   p = &age;
   q = &average;
}
```



Assignment Operator =

- Like `int` and `float` variables, a pointer can be assigned, if they are compatible (pointing to the same type).

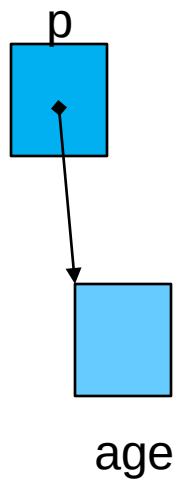
```
int main()
{
    int age;
    float average;
    int *p, *q;
    float *r, *s;

    p = &age;
    q = p;
    r = &average;
    s = r;

    p = &average; // error
    q = r; // error
}
```



Dereferencing Operator *

- What's the purpose of storing addresses in pointers?
- We can access the pointee through the pointer by the dereferencing operator.

```
int main()
{
    int age = 30;
    int *p;

    p = &age;
    printf("age: %d\n", *p); // age: 30

    *p = 45;

    printf("age: %d\n", age); // age: 45

    age = *p+2;
    printf("age: %d\n", age); // age: 47
}
```



Dereferencing Operator *

Operators	Associativity	Type
() []	left to right	highest
+ - ++ -- ! * & (*type*)	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > s>=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

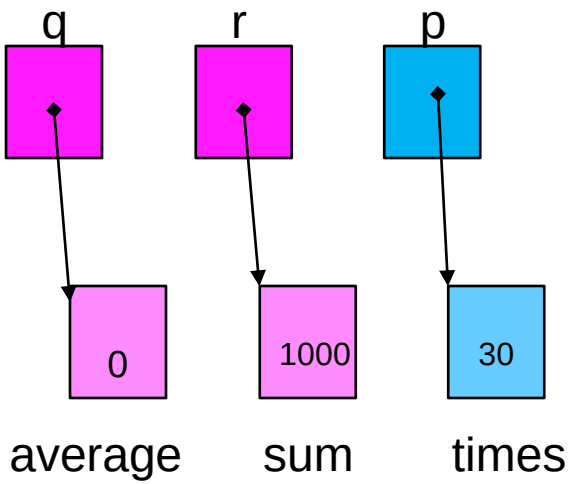
Fig. 7.5 | Operator precedence and associativity.

Deitel and Deitel, *C How to Program*, 6th ed.

Dereferencing Operator *

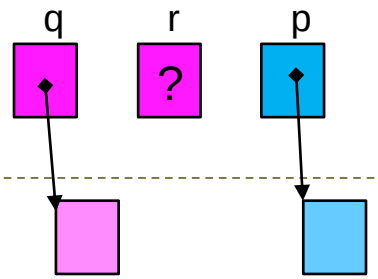
- More examples

```
int main()
{
    int times = 30;
    float sum = 1000, average = 0;
    int *p;
    float *q, *r;

    p = &times;
    q = &average;
    r = &sum;

    *q = (*r)/(*p); // average = sum/times;
    printf("%f", average);
}
```



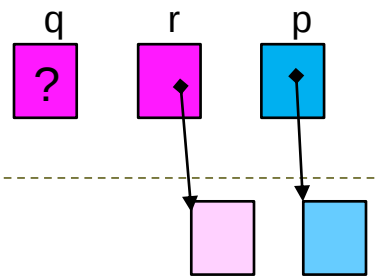
Dereferencing Operator *

- After learning pointers, your programs crash more often (since you usually forget to set proper values to pointers).




```
int main()
{
    int times = 30;
    float sum = 1000, average = 0;
    int *p;
    float *q, *r;

    p = &times;
    q = &average;

    *q = (*r)/(*p);
    printf("%f", average);
}
```

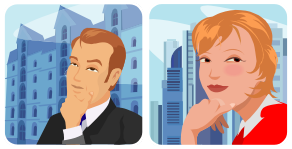
This usually produces wrong values only.




```
int main()
{
    int times = 30;
    float sum = 1000, average = 0;
    int *p;
    float *q, *r;

    p = &times;
    r = &sum;

    *q = (*r)/(*p);
    printf("%f", average);
}
```

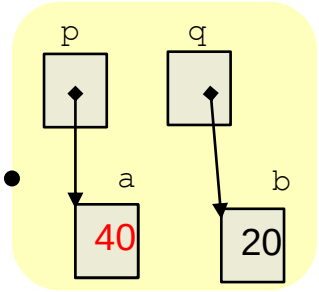
This usually crashes.

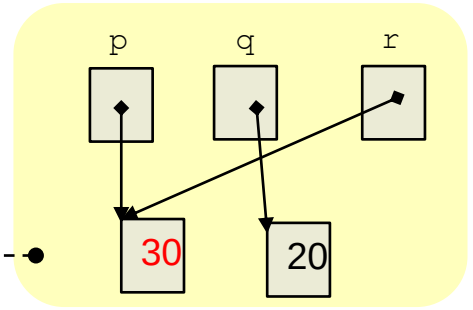
Exercise

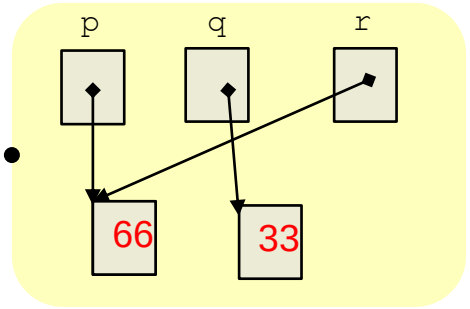


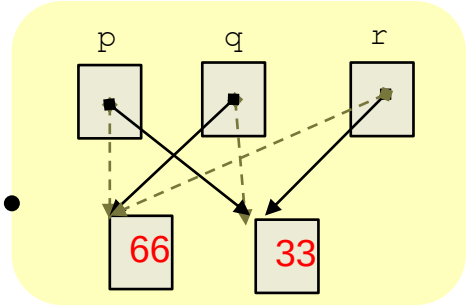
```
int main()
{
    int a = 10, b = 20;
    int *p, *q, *r;

    p = &a;
    q = &b;

    *p = *q + b;
    print(a, b);

    r = p;
    a = 30;
    print(*q, *r);

    *r += 3;
    *q = *r;
    print(a, b);

    *p = *q + *r;
    print(a, b);

    print(*q, *r);
    p = q;
    q = r;
    r = p;
    print(*q, *r);

}
```

```
void print(int a, int b)
{
    printf("%d %d\n", a, b);
}
```









40 20
20 30
33 33
66 33
33 66
66 33

Initialization

- Now the asterisk (*) has three different meanings in the program.
  - Multiplication operator
  - Definition of a pointer
  - Dereferencing operator

```
int main()
{
    int x, y;
    int *p;

    p = &x;

    y = x * 20;
    *p = y + 100;
}
```

Initialization

- Don’t get confused between the 2nd and 3rd meanings.
  - Multiplication operator
  - Definition of a pointer
  - Dereferencing operator

```
int main()
{
    int x;
    int *p = &x; // initialize p with &x
    int *q = 100; // error

    p = &x; // assign &x to p
    p = 100; // error
    *p = 100; // assign 100 to what p points
    *p = &x; // error
}
```

Initialization

- To avoid dereferencing an uninitialized pointer, we need a particular value to indicate the “un-initialized” state.
- We set the null pointer by `0`.
  - 0 is the only integral value that can be assigned to a pointer.
  - In C, `NULL` is often used. However, `0` is common in C++.
  - Since C++11, `nullptr` is introduced.

```
int main()
{
   int x = 0;
   int *p = 0; // set p to be a null pointer

   // p = &x;

   if (p != 0) // a safety check
   {
      *p = 100;
   }
}
```

```
int main()
{
   int x = 0;
   int *p = nullptr;

   // p = &x;

   if (p != nullptr)
   {
      *p = 100;
   }
}
```

Relational operators

- Relational operators are applicable to pointers.
  - Note that we are comparing the values of pointers (the addresses), not the values of the pointees.
  - The equality (==) and inequality (!=) operator are much often used than the others.

```
int *search(const int data[], int size, int key)
{
    // ...
    // if not found, return nullptr
    return nullptr;
}
int main()
{
    int arr[5] = {7, 8, 4, 2, 9}, key = 9;

    int *pos = search(arr, 5, key);
    if (pos != nullptr)
    {
        // do something
    }
}
```

```
int main()
{
    int arr[5] = {};
    int *p = &arr[0], *q = &arr[2];

    if (q > p)
    {
        printf("Yes, q is behind p.\n");
    }
}
```

Review of Mentioned C++ Features

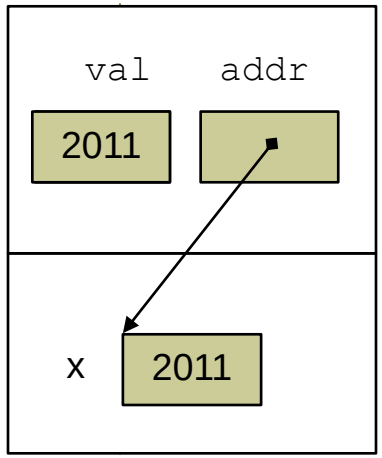
- Although we still kept using `printf()` and `scanf()` in our programs, we are learning C++.
  - As I mentioned earlier, we use `printf()` and `scanf()` in this course since many concepts related to `std::cin` and `std::cout` are not mentioned in this course.

	C++	C
`bool`	`bool` is a built-in type.	`bool` is simulated by the built-in type `_Bool` and the header `<stdbool.h>`.
`constexpr`	`constexpr` is a keyword.	`const` ☑ `constexpr` ☒
array initialization	`int arr[5] = {};` `int arr[5] = {0};`	`int arr[5] = {};` ☒ `int arr[5] = {0};` ☑
VLA	C++ does not allow VLA.	`int n = 3;` `int arr[n];`
function overloading	`int max(int, int);` `int max(int, int, int);`	`int max2(int, int);` `int max3(int, int, int);`
nullptr	`int *p = nullptr;`	`int *p = NULL;` `int *q = 0;`

Passing Addresses to Functions

- We know how to pass an integer or a floating-point value to a function.
- What about passing an address?

```
void foo(int val, int *addr)
{
    printf("Value = %d, address = %p\n", val, addr);
}

int main()
{
    int x = 2011;

    foo(x, &x);

    printf("x = %d, &x = %p\n", x, &x);
}
```



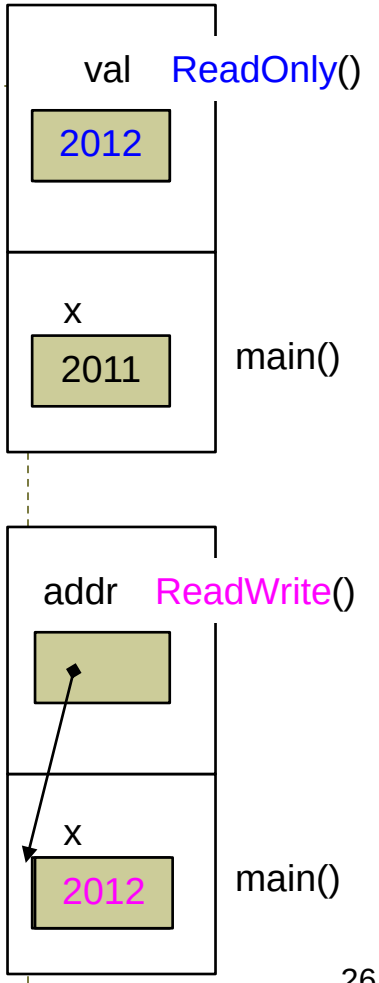
Value = 2024, address = 000000000061FE1C
x = 2024, &x = 000000000061FE1C

Passing Addresses to Functions

- With the address, I can do anything (both read or write) I want.

```
void ReadOnly(int val)
{
   printf("Value = %d\n", val);
   val = 2012;
}

void ReadWrite(int *addr)
{
   printf("Value = %d\n", *addr);
   *addr = 2012;
}


int main()
{
   int x = 2011;

   ReadOnly(x);
   printf("After ReadOnly(), x = %d\n", x);

   ReadWrite(&x);
   printf("After ReadWrite(), x = %d\n", x);
}
```

```
Value = 2011
After ReadOnly(), x = 2011
Value = 2011
After ReadWrite(), x = 2012
```



Passing Addresses to Functions

- Pass by value or pass by (value of) address?
  - Pass by value when the passed value is only read in the called function.

```
printf("%d", x);
```

  - Pass by address when the passed value will be modified in the called function.

```
scanf("%d", &x);
```

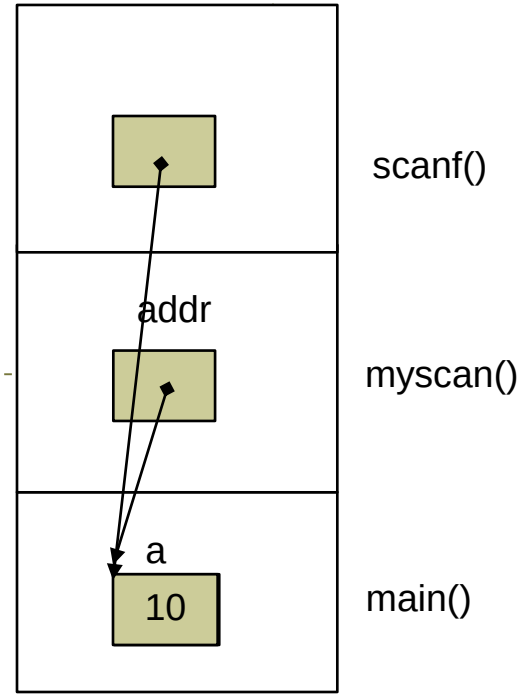
Now you know the meaning and purpose of the mysterious `&` in the call of `scanf()`.

  - Technically, pass-by-address is just a special case of pass-by-value, where the passed value is an address.

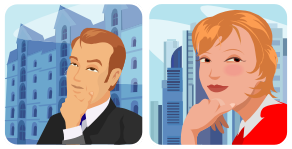
Passing Addresses to Functions

- `&` is not always necessary in `scanf()`.

```
void myscan(int *addr)
{
    printf("Please input an integer...>");
    scanf("%d",  addr  );
}

int main()
{
    int a = 10;
    myscan(&a);
    printf("Your input is %d.\n", a);
}
```



Exercise

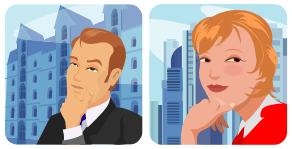


```
int Global_1 = 1;
void func1(int val) { /* Code segment A */ }
void func2(int *ptr) { /* Code segment B */ }
int Global_2 = 2;

int main()
{
    /* Code segment C */
    int local_1 = 3;
    /* Code segment D */
    func1(local_1);
    func2(&local_1);
    /* Code segment E */
    int local_2 = 4;
    func1(local_2);
    func1(Global_2);
    func2(&Global_2);
    /* Code segment F */
    if (local_1 > 0)
    {
        int local_3 = 5;
        /* Code segment G */
    }
}
```

Check value here – Which code segments should you check if the value is modified?

Global_1: A~G
Global_2: B~G
local_1: B, D~G
local_2: F, G
local_3: G

Exercise



```
void reset(int *p)
{
    *p = 0;
}
// ---------------------------------
void calc(int x, int y, int *z)
{
    x += 1;
    y += 1;
    *z = x+y;
}
// ---------------------------------
bool divide(int x, int y, int *q, int *r)
{
    if (y == 0) return false;

    *q = x/y;
    *r = x%y;

    return true;
}
// ---------------------------------
void print(int a, int b)
{
    printf("%d %d\n", a, b);
}
```

```
int main()
{
    int a = 10, b = 20;

    calc(11, 12, &a);
    print(a, b);

    if (divide(a, b, &a, &b))
    {
        print(a, b);
    }

    reset(&b);

    if (divide(a, b, &a, &b))
    {
        print(a, b);
    }
}
```

```
25 20
1 5
```

Passing Addresses to Functions

- Return by value or pass by address?
  - When you want to modify a single value, you can choose to return by value or pass by address.

```
int sum1(int a, int b)
{
    return a+b;
}
void sum2(int a, int b, int *ans)
{
    *ans = a+b;
}
int main()
{
    int x = 0;

    x = sum1(10, 20);

    sum2(10, 20, &x);
}
```

Passing Addresses to Functions

- Return by value or pass by address?
  - In most cases, return-by-value is more convenient.

convenient

```
int year = 0;

// input ...

if (isLeapYear(year))
{
      // do something special
}
```

inconvenient

```
int year = 0, check = 0;

// input ...

isLeapYear(year, &check);

if (check)
{
      // do something special
}
```

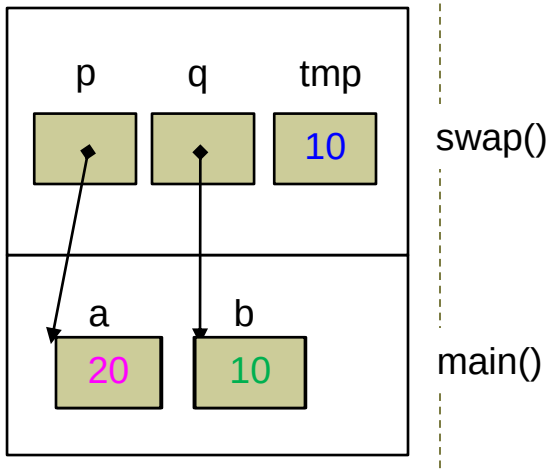
Passing Addresses to Functions

- Return by value or pass by address?
  - When you have to modify more than one variable, you need to pass by address(es).
  - Now you can complete the `swap()` function.

Exercise: `swap()`

- A correct implementation

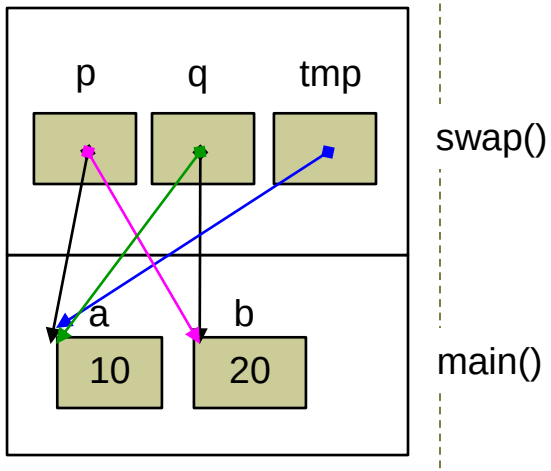
```
void swap(int *p, int *q)
{
    int tmp = *p;
    *p = *q;
    *q = tmp;
}

int main()
{
   int a = 10, b = 20;
   printf("a = %d, b = %d\n", a, b);

   swap(&a, &b);

   printf("a = %d, b = %d\n", a, b);
}
```



Exercise: `swap()`

- Common incorrect implementation 1

```
void swap(int *p, int *q)
{
    int *tmp = p;
    p = q;
    q = tmp;
}

int main()
{
   int a = 10, b = 20;
   printf("a = %d, b = %d\n", a, b);

   swap(&a, &b);

   printf("a = %d, b = %d\n", a, b);
}
```



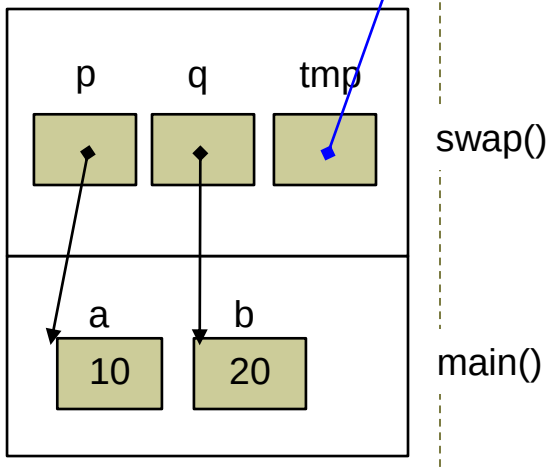
Exercise: `swap()`

- Common incorrect implementation 2

```
void swap(int *p, int *q)
{
    int *tmp;
    *tmp = *p;
    *p = *q;
    *q = *tmp;
}

int main()
{
   int a = 10, b = 20;
   printf("a = %d, b = %d\n", a, b);

   swap(&a, &b);

   printf("a = %d, b = %d\n", a, b);
}
```

Sometimes it works (mistakenly), but you should know where the problem is.

??



Exercise: Winner Takes All

```
#include <stdio.h>

void winner_take_all(int *pa, int *pb, int *pc)
{
    int sum = *pa + *pb + *pc;

    int *w = nullptr;
    if (*pa >= *pb && *pa >= *pc) { w = pa; }
    else if (*pb >= *pa && *pb >= *pc) { w = pb; }
    else { w = pc; }

    *pa = *pb = *pc = 0;
    *w = sum;
}

int main()
{
    int a = 0, b = 0, c = 0;

    scanf("%d%d%d", &a, &b, &c);   // 100 150 20
    winner_take_all(&a, &b, &c);
    printf("%d %d %d\n", a, b, c); // 0 270 0
}
```

Exercise: Winner Doubles

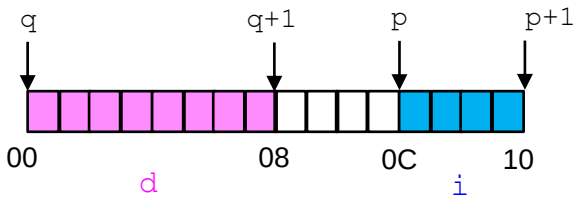
```c
#include <stdio.h>

int *winner(int *pa, int *pb, int *pc)
{
    if (*pa >= *pb && *pa >= *pc) { return pa; }
    else if (*pb >= *pa && *pb >= *pc) { return pb; }
    else { return pc; }
}

int main()
{
    int a = 0, b = 0, c = 0; // 彩票數量
    scanf("%d%d%d", &a, &b, &c); // 輸入原始彩票數量
    int *w = winner(&a, &b, &c); // 找出贏家
    *w *= 2; // 贏家拿雙倍
    printf("%d %d %d\n", a, b, c); // 列印最終的彩票數量
 }
```

Operator +/-

- We can add an integer to or subtract an integer from an address.
- `Addr ± i` means to increase/decrease *i* units of the associated data type.

```
int main()
{
    int i = 0;
    double d = 0;
    int *p = &i;
    double *q = &d;

    printf("Each int occupies %zu bytes.\n", sizeof(int));
    printf("p = %p, p+1 = %p, p-1 = %p.\n", p, p+1, p-1);
    printf("Each double occupies %zu bytes.\n", sizeof(double));
    printf("q = %p, q+1 = %p, q-1 = %p.\n", q, q+1, q-1);

    printf("p+2 = %p.\n", p+2);
}
```

Each int occupies **4** bytes.
p = 000000000061FE0C, p+1 = 000000000061FE10, p-1 = 000000000061FE08.
Each double occupies 8 bytes.
q = 000000000061FE00, q+1 = 000000000061FE08, q-1 = 000000000061FDF8.
p+2 = 000000000061FE14.

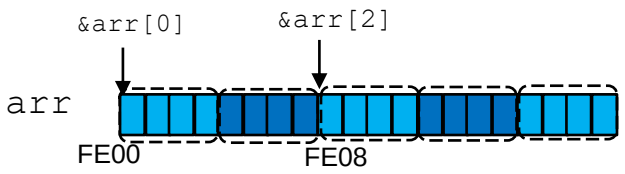


Operator +/-

- We cannot add two pointers, but we can subtract them.
  - These two pointers must be the same type.
- The result is the number of associated units between the two pointers.

```
int main()
{
   int arr[5] = {};

   printf("arr[0]: %p, arr[2]: %p.\n", &arr[0], &arr[2]);
   printf("%td", &arr[2] - &arr[0]);
}
```

```
arr[0]: 000000000061FE00, arr[2]: 000000000061FE08.
2
```



Notes.

`%p` expects pointers of `void *`, but here we just let it go.

```
printf("%p", static_cast<void *>(&arr[0]));
```

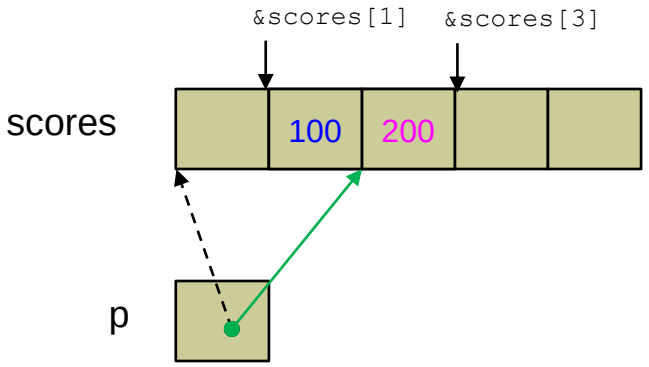
Subtraction is only valid when the two pointers point to the same array; otherwise, it causes an undefined behavior.

Operator +/-

- operator +/- is useful only when accessing an array.
  - The elements in an array occupy contiguous memory spaces.

```
int main()
{
    int scores[5] = {};

    int *p = &scores[0];

    *(p+1) = 100;

    p += 2;
    *p = 200;

    for (int i=0; i<5; i+=1)
    {
        printf("%d ", scores[i]);
    }

    printf("\nDifference: %d", &scores[3] - &scores[1]);
}
```

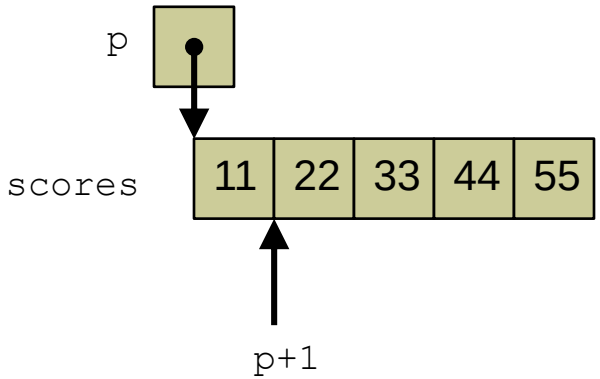
```
0 100 200 0 0
Difference: 2
```



Operator []

- `x[i]` is just a simplified form of `*(x+i)`.

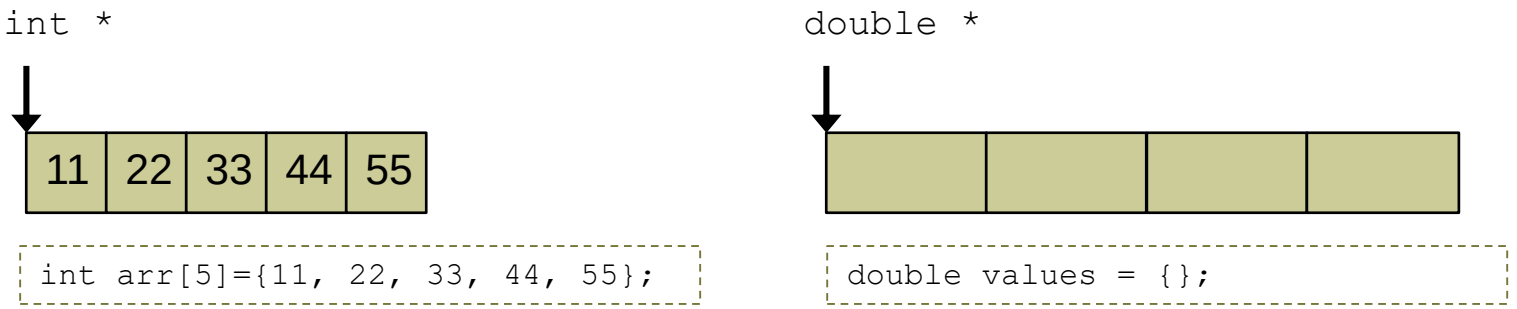
```
int main()
{
    int scores[5] = {11, 22, 33, 44, 55};

    int *p = &scores[0];

    for (int i=0; i<5; i+=1) {
        printf("%d ", *(p+i));
    }
    printf("\n");

    for (int i=0; i<5; i+=1) {
        printf("%d ", p[i]);
    }
    printf("\n");

    for (int i=0; i<5; i+=1) {
        printf("%d ", *p);
        p += 1;
    }
    printf("\n");

}
```




```
11 22 33 44 55
11 22 33 44 55
11 22 33 44 55
```

Operator []

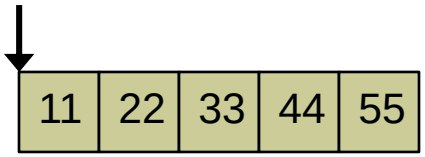
- Array-to-pointer conversion
  - When we want to treat an array as a single value in the assignment =, dereference *, arithmetic +/−, and relational operations (e.g. >), there is an implicit *array-to-pointer conversion*.
  - After the conversion, we get an address:
    - Its *type* is a pointer to the element of the array.
    - Its *value* is the address of the first element in the array.



Operator []

- Array-to-pointer conversion

```
int main()
{
   int scores[5] = {11, 22, 33, 44, 55};

   printf("The address of the 1st element in scores is %p.\n", scores); // same as &scores[0]
   printf("The value of the 1st element in scores is %d.\n", *scores); // same as scores[0]
   printf("The value of the 2nd and 3rd elements are %d and %d.\n", *(scores+1), scores[2]);

   int *p = scores;

   printf("p = %p, *p = %d.\n", p, *p);

   if (p == scores)
   {
      printf("Yes, now p points to the 1st element in scores.\n");
   }

   p = scores+1;
   printf("p = %p, *p = %d.\n", p, *p);

}
```




```
The address of the 1st element in scores is 000000000061FE00.
The value of the 1st element in scores is 11.
The value of the 2nd and 3rd elements are 22 and 33.
p = 000000000061FE00, *p = 11.
Yes, now p points to the 1st element in scores.
p = 000000000061FE04, *p = 22.
```

Operator []

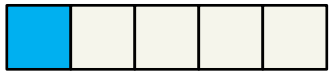
- Now you know why `arr[0]` refers to the first element of the array `arr`.
  - `arr[0]` is equal to `*(arr+0)`.
  - There is an implicit array-to-pointer conversion when we do `arr+0`.
  - The converted address points to the first element of the array.
  - Adding `0` keeps the address unchanged.
  - Dereferencing the pointer gets the pointed variable, that is, the first element.

`arr[0]` ≡ `*(arr+0)`

Three steps

① array-to-pointer conversion: `int [5]` → `int *`
② pointer addition (`+`): `int *` → `int *`
③ dereference (`*`): `int *` → `int`

`int arr[5];`



I Don’t Owe You Now.

```
#include <stdio.h>

int main()
{
   int key = 0;

   printf("Please input the value to search...>");
   scanf("%d", &key);

   int data[5] = {11, 22, 33, 44, 55};

   bool found = false;
   for (int i=0; i<5; i+=1)
   {
       if (data[i] == key)
       {
           printf("The value %d is found in data[%d].\n", key, i);
           found = true;
           break;
       }
   }

   if (!found)
   {
       printf("Data not found!");
   }

   return 0;
}
```

Everything is explained.