Review of Pointers and References

Instructor: Tsung-Che Chiang
tcchiang@ieee.org
Department of Computer Science and Information Engineering
National Taiwan Normal University
https://moodle3.ntnu.edu.tw/course/view.php?id=33738

Definitions and Operators

- 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;
}
```

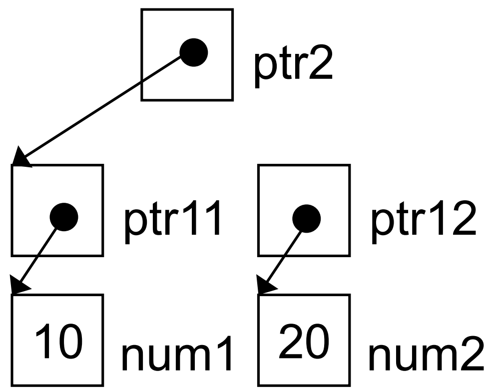
Initialization

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

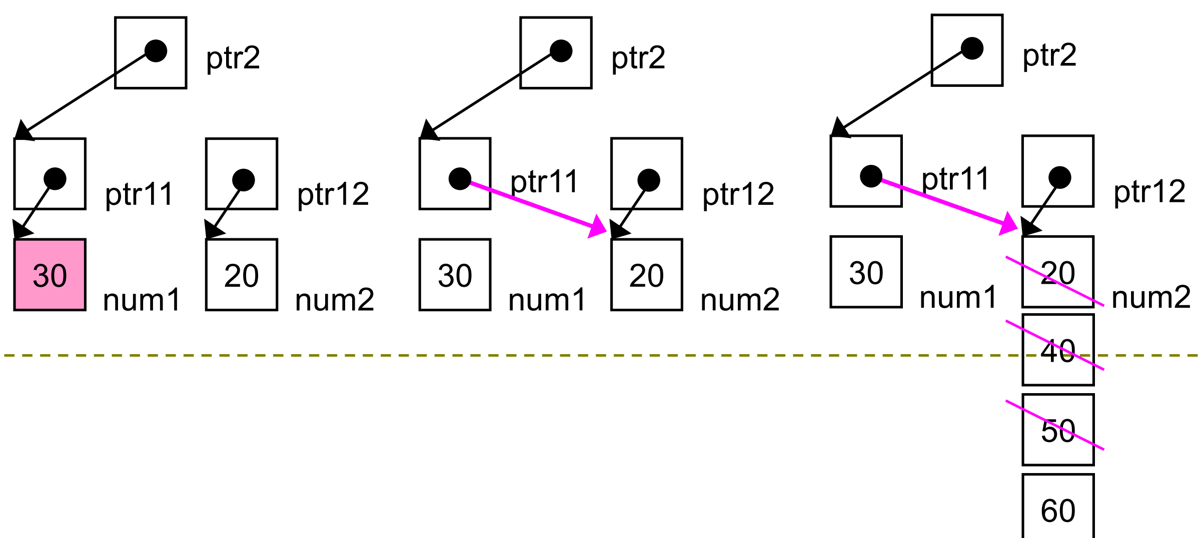
```
int main()
{
    int x;
    int *p = &x; // initialize p with &x
    int *q = 100; // error

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

Pointers & Individual Variables

```
int main()
{
   int num1 = 10, num2 = 20,
       *ptr11, *ptr12,
       **ptr2;

   ptr11 = &num1;
   ptr12 = &num2;
   ptr2 = &ptr11;
```

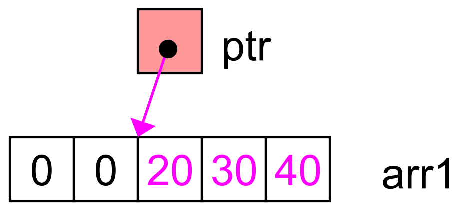


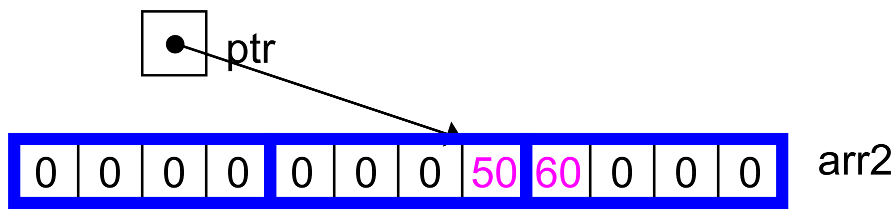

```
   *ptr11 = 30;
   *ptr2 = &num2;
   *ptr11 = 40;
   *ptr12 = 50;
   **ptr2 = 60;
}
```



Pointers and Arrays

```
int main()
{
    int arr1[5]={}, arr2[3][4]={},
        *ptr=0;

    ptr = &arr1[2];
    *ptr = 20;
    *(ptr+1) = 30;
    ptr[2] = 40;



    ptr = &arr2[1][3];
    *ptr = 50;
    ptr[1] = 60;


}
```





Operator []

- Why does `arr[0]` refer 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 want to 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 of the array.

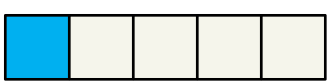
So someone may tell you that you can write `a[0]` as `0[a]`.
Don't do that unless you want to make your code unreadable.

`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];`



Array vs. Pointer

Misconception: "array = pointer"

- They said that an array is a pointer just because the following two loops produce the same output.

```
#include <stdio.h>
int main()
{
    int arr[5]={2, 4, 7, 8, 9};
    int *ptr = arr;

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

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

```
2 4 7 8 9
2 4 7 8 9
```

Note that the coding style of this example program is not good – we write the data size by three duplicate "magic" numbers (5).
In addition, we will keep using C-style I/O `printf()/scanf()` before we talk about operator overloading and introduce `std::cin/cout` then.

Array vs. Pointer

Misconception: "array = pointer"

- If an array is a pointer, why can't we do this?
  - Actually, the error message already tells you that an array is not a pointer.

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

```
error: incompatible types in assignment of 'int*' to 'int [5]'
```

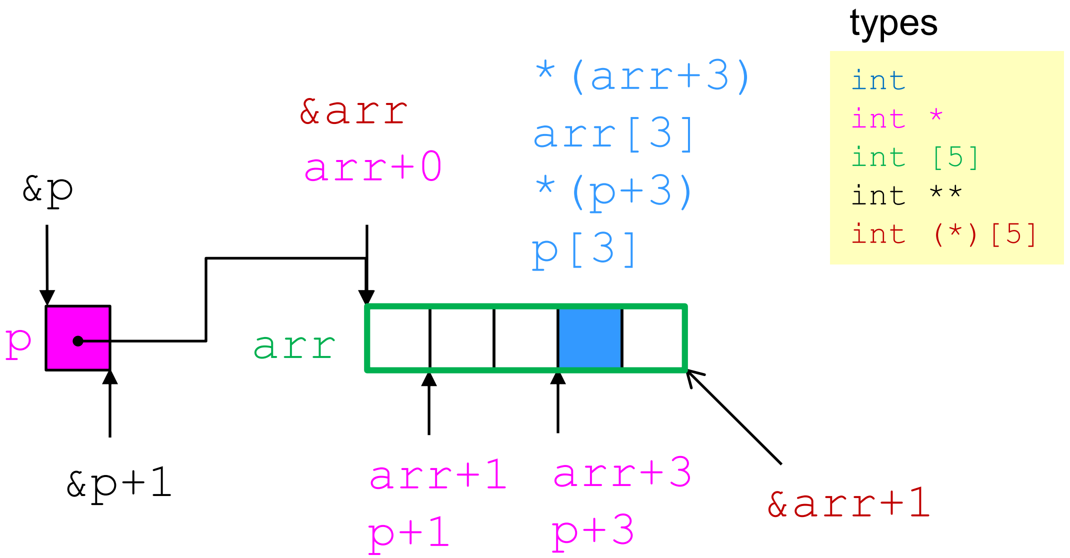
- Okay. Then, they turned to say that an array is a constant pointer. But, why do they have different sizes?

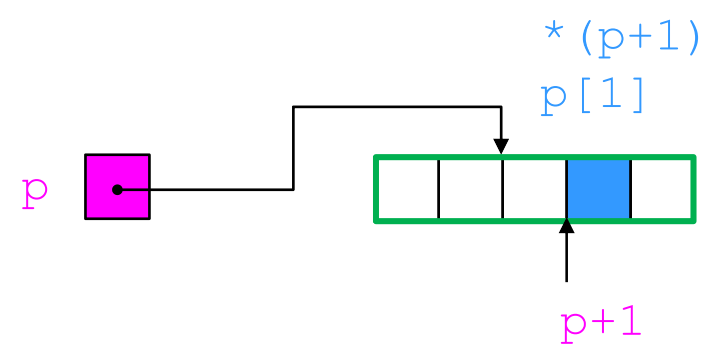
```
int main()
{
    int arr[5] = {};
    int *p = 0;
    if (sizeof(arr)!=sizeof(p))
        puts("Unequal size");
}
```

```
Unequal size
```

Array vs. Pointer

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

    p = arr+2;

}
```





Review: The Five Rules

Rule 1: (Implicit conversion)

If *A* is an array of objects of type T,
the implicitly converted address points to T.

```
int num[5];    // num is an array of int
int *p = num;  // the converted address is of type int *

int *aop[3];   // aop is an array of int *
int **pp = aop;// the converted address is of type int **

int tab[4][3]; // tab is an array of int [3]
int (*p2a)[3] = tab; // the converted address is of type int (*)[3]
```

Review: The Five Rules

- Rule 2: (Address-of)

  If *A* is of type T,
  &*A* is a pointer to T (i.e. &*A* is of type T*).

```
int num;        // num is int
int *p = &num; // &num is int *

int *p;         // p is int *
int **pp = &p; //  &p is int **

int arr[3];     // arr is int [3]
int (*p2a)[3] = &arr; // &arr is int (*)[3]
```

Review: The Five Rules

- Rule 3: (Dereference)

  If *A* is of T*, **A* is of type T.

```
int m = 0, n = 0, arr[3];

int *p = &m;    // p is int *
*p = n;         // *p is int

int **pp = &p;  //  pp is int **
*pp = &n;       //  *pp is int *

int (*p2a)[3] = &arr; // p2a is int (*)[3]
(*p2a)[2] = m;        // (*p2a) is int [3]
```

Review: The Five Rules

- Rule 4: (Address shift)

  If *A* is of type T*,
  A+*x* moves forward by *x* objects of type T,
  i.e. moving *x*·`sizeof(T)` bytes.

```
int *p;    // p is of type int *
p+1        // moves 1 int

int **pp;  // pp is of type int **
pp+1       // moves 1 int *

int (*p2a)[3]; // p2a is of type int (*)[3]
p2a+1          // moves 1 int [3]
```

Review: The Five Rules

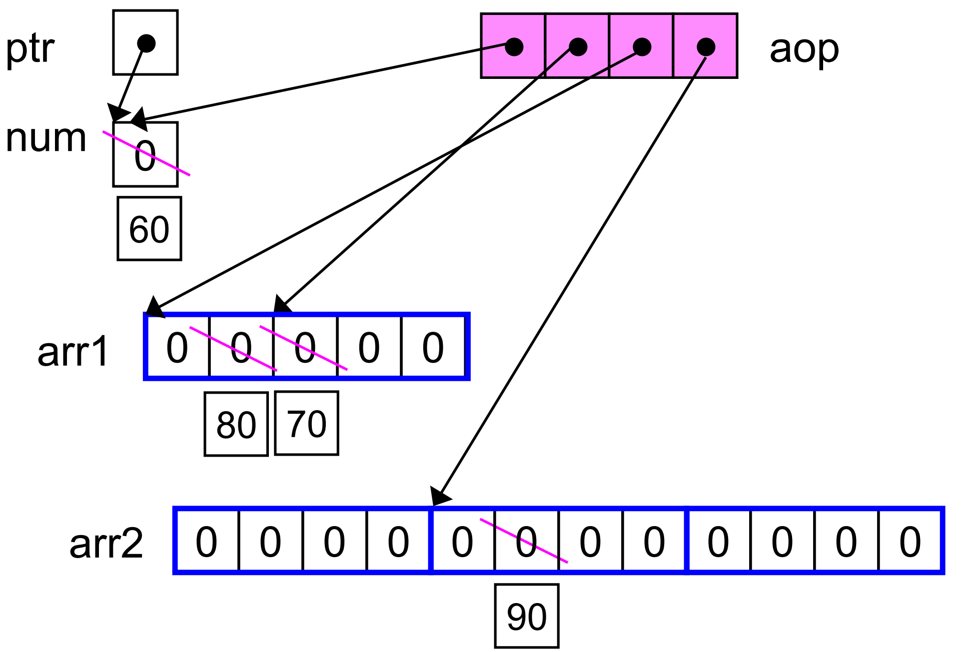
- Rule 5: (operator [])

$$A[x] \equiv *(A+x)$$

Recall Rule 1, 4, and 3.

Array of Pointers

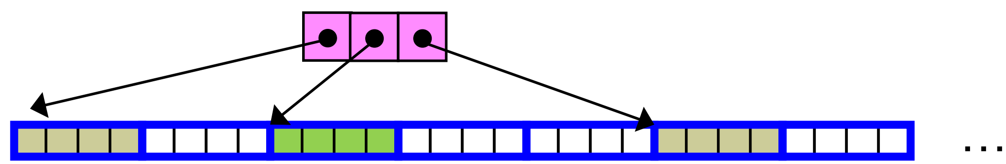
```
int main()
{
    int num=0, arr1[5]={}, arr2[3][4]={},
        *ptr=0,
        *aop[4]={};

    ptr = &num;
    aop[0] = ptr;
    *aop[0] = 60;

    aop[1] = &arr1[2];
    *aop[1] = 70;

    aop[2] = arr1;
    aop[2][1] = 80;

    aop[3] = arr2[1];
    aop[3][1] = 90;
}
```

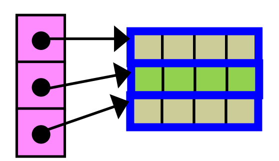


Array of Pointers

- It can extract a 2-D array "virtually".

```
int main()
{
    int arr[10][4]={},
        *aop[3]={};

    aop[0] = arr[0];
    aop[1] = arr[2];
    aop[2] = arr[5];

    for (int i=0; i<3; i+=1)
    {
        for (int j=0; j<4; j+=1)
        {
            printf("%d ", aop[i][j]);
        }
        printf("\n");
    }
}
```



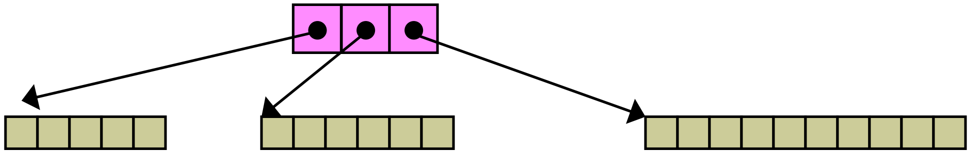


Array of Pointers

- The 2-D (virtual) array can be composed of variable-length 1-D arrays.

```
int main()
{
   int arr1[5]={7}, arr2[6]={8}, arr3[10]={9},
       *aop[3]={arr1, arr2, arr3}, sizes[3]={5, 6, 10};

   for (int i=0; i<3; i+=1) {
      for (int j=0; j<sizes[i]; j+=1) {
         printf("%d ", aop[i][j]);
      }
      printf("\n");
   }
}
```

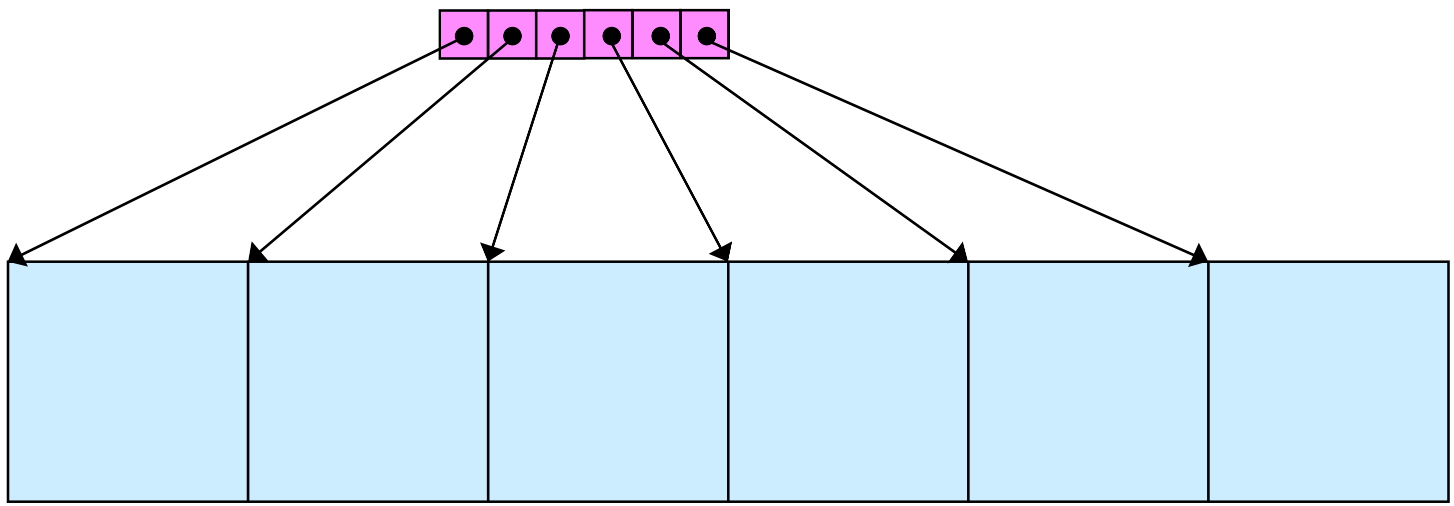
```
7 0 0 0 0
8 0 0 0 0 0
9 0 0 0 0 0 0 0 0 0
```



To make the code more maintainable, we can use `std::size()`:
`int sizes[] = {std::size(arr1), std::size(arr2), std::size(arr3)};`

Array of Pointers

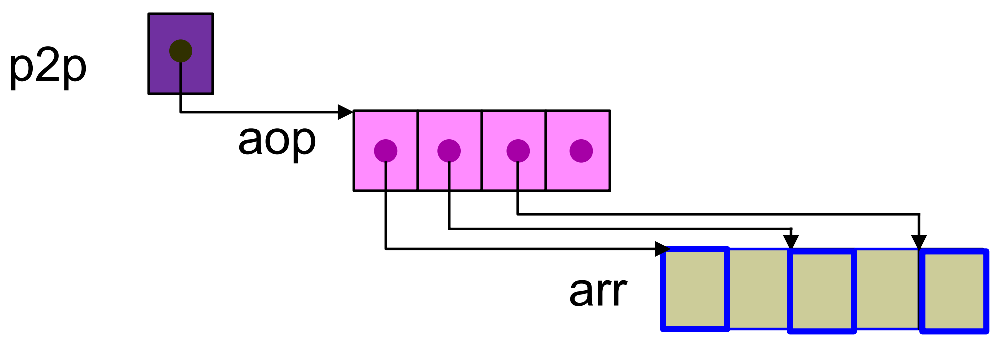
- Another common usage of the array of pointers is to sort an array of "big" objects indirectly but "efficiently".



Array of Pointers

- Note the difference between a pointer to pointer and an array of pointers.
  - Remember the difference between a pointer (e.g. `int *p`) and an array (e.g. `int a[5]`)?

```
int arr[5] = {};  // an array of 5 integers

int *aop[4] = {}; // an array of 4 pointers to int
aop[0] = arr;
aop[1] = &arr[2];

int **p2p = aop;  // a pointer to pointer to int
p2p[2] = &arr[4];
```

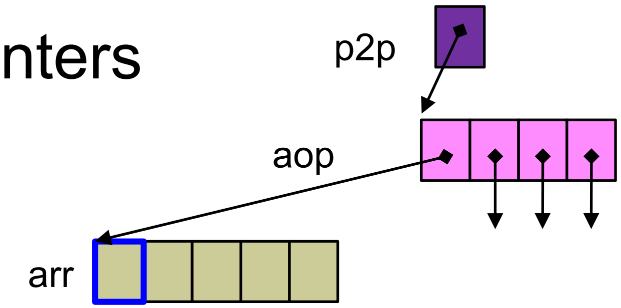


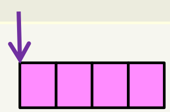
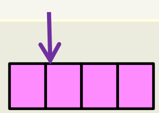
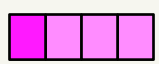
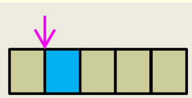
Recall the five rules!

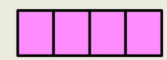
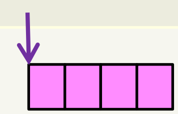
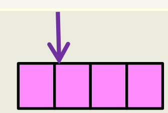
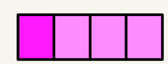
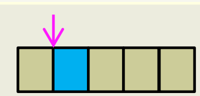
Array of Pointers

- Pointer to pointer vs. array of pointers

```
int arr[5] = {};
int *aop[4] = {}; // an array of 4 pointers to int
aop[0] = arr;
int **p2p = aop; // a pointer to pointer to int
```



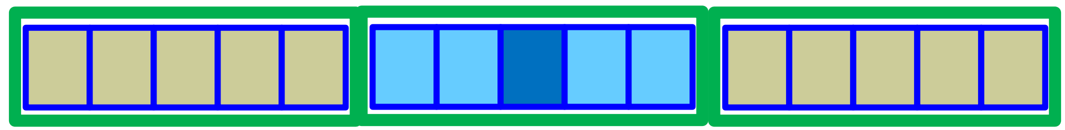
Expression	Type	Meaning
p2p	int **	
p2p+0	int **	
p2p+1	int **	
p2p[0]	int *	
p2p[0]+1	int *	
p2p[0][1]	int	

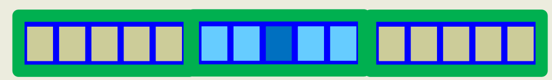
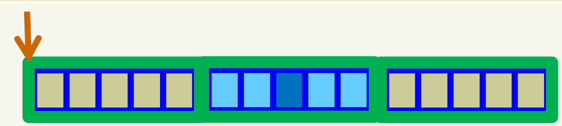
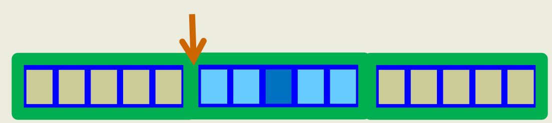
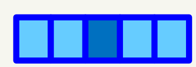
Expression	Type	Meaning
aop	int *[4]	
aop+0	int **	
aop+1	int **	
aop[0]	int *	
aop[0]+1	int *	
aop[0][1]	int	

Pointer to Array

- Access of an element in a 2-D array

```
int main()
{
    int arr[3][5];
}
```

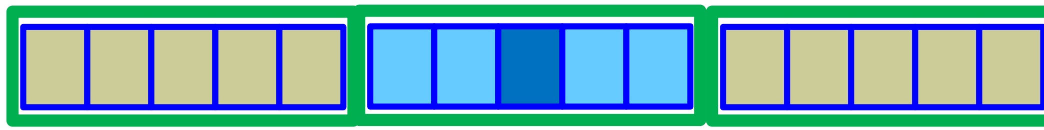


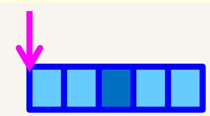
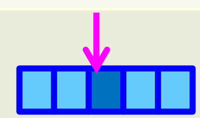
Expression	Type	Meaning
arr	int [3][5]	
arr+0	int (*)[5]	
arr+1	int (*)[5]	
arr[1] ≡ *(arr+1)	int [5]	

Pointer to Array

- Access of an element in a 2-D array

```
int main()
{
    int arr[3][5];
}
```

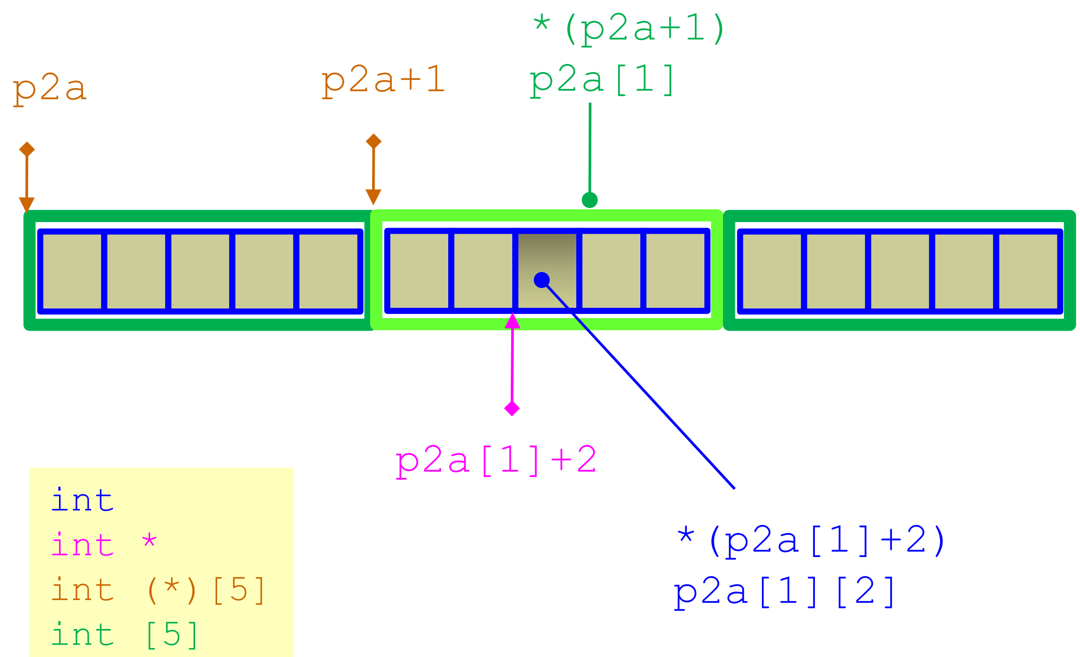


Expression	Type	Meaning
arr[1]	int [5]	
arr[1]+0	int *	
arr[1]+2	int *	
arr[1][2] ≡ *(arr[1]+2) ≡ *(*(arr+1)+2)	int	

Pointer to Array

- Recall the
  - array-to-pointer conversion
  - arithmetic operations of pointers
  - meaning of operator `[]`, `x[i]` ≡ `*(x+i)`

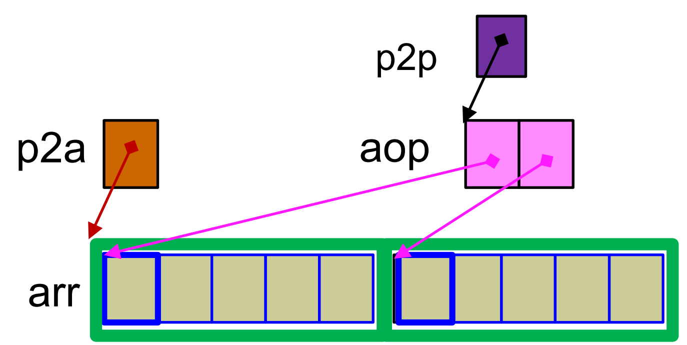
```
int main()
{
    int TwoDim[3][5] = {};

    int (*p2a)[5] = TwoDim;

    for (int i=0; i<3; i+=1)
    {
        for (int j=0; j<5; j+=1)
        {
            printf("%d ", p2a[i][j]);
        }
        printf("\n");
    }
}
```



Pointer to Array

- Note the difference between a pointer to array, an array of pointers, and a pointer to pointer.
  - Do not think they are equal just because you can put `arr`, `p2a`, `aop`, or `p2p` in the blank ___ to produce the same output.

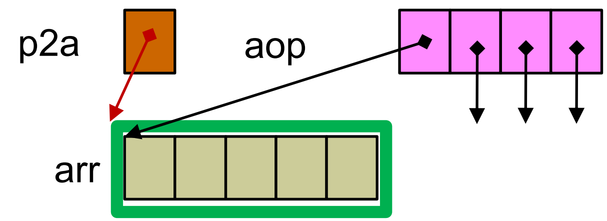
```
int main()
{
    int arr[2][5] = {{1,3,7}, {9,8,2}};
    int (*p2a)[5] = arr; // p2a: a pointer to an array of 5 ints
    int *aop[2] = {arr[0], arr[1]}; // aop: an array of 2 pointers to int
    int **p2p = aop; // p2p: a pointer to pointer to int


    for (int i=0; i<2; i+=1)
    {
        for (int j=0; j<5; j+=1)
        {
            printf("%d ", ___[i][j]);
        }
    }
}
```



Pointer to Array

- Pointer to array vs. array of pointers

```
int arr[5] = {};
int (*p2a)[5] = &arr; // a pointer to an array of 5 ints

int *aop[4] = {}; // an array of 4 pointers to int
aop[0] = arr;
```



Expression	Type	Meaning
p2a	int (*)[5]	
p2a+0	int (*)[5]	
p2a+1	int (*)[5]	
p2a[0]	int [5]	
p2a[0]+1	int *	
p2a[0][1]	int	

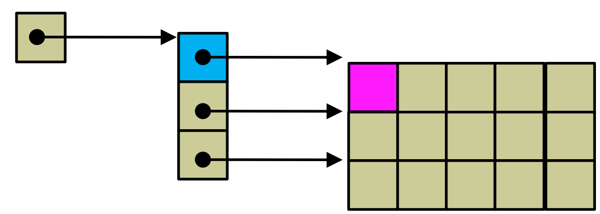
Expression	Type	Meaning
aop	int *[4]	
aop+0	int **	
aop+1	int **	
aop[0]	int *	
aop[0]+1	int *	
aop[0][1]	int	

Array vs. Pointer

Misconception: "2-D array = pointer to pointer"

- They thought that a 2-D array is a thing like this:

```
int arr[3][5]={};
```



- If a 2-D array is a pointer to pointer, why can't this work?

```
int arr[3][5]={};
int **p2p = arr;
```

error: cannot convert 'int (*)[5]' to 'int**' in initialization

- If a 2-D array is the thing shown above, how could the two addresses be the same?

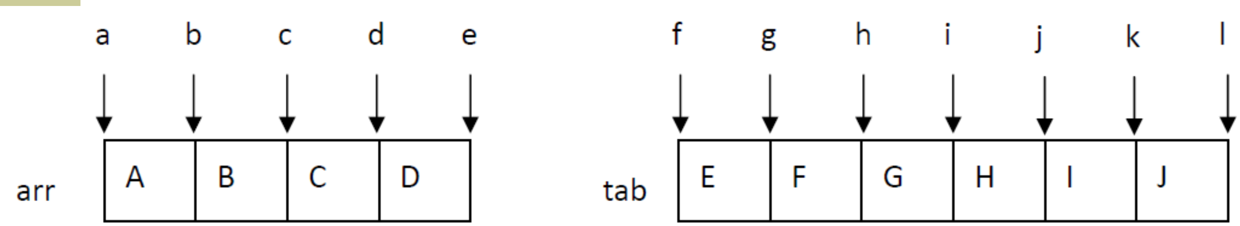
```
int arr[3][5]={};
printf("%p %p\n", &arr[0], &arr[0][0]);
```

Types of Pointers and Arrays

```
int main()
{
    int num,            // an integer
        arr[5],         // an array of 5 integers
        arr2[3][5],     // an array of 3 arrays of 5 ints
        *ptr1,          // a pointer to integer
        **ptr2,         // a pointer to pointer to integer
        (*ptr3)[5];     // a pointer to array of 5 integers

    ptr1 = num;     ptr1 = &num;     // *ptr1 refers to num
    ptr1 = arr;     ptr1 = arr2;     // ptr1[i] refers to arr[i]

    ptr2 = num;     ptr2 = &num;
    ptr2 = &ptr1;   ptr2 = arr;      // *ptr2 refers to ptr1
    ptr2 = arr2;

    ptr3 = num;     ptr3 = &num;
    ptr3 = &ptr1;   ptr3 = &ptr2;
    ptr3 = &arr;                     // ptr3[0][i] refers to arr[i]
    ptr3 = arr2;                     // ptr3[i][j] refers to arr2[i][j]
    ptr3 = &arr2;
}
```

Exercise

```
int main()
{
    int arr[4];
    int tab[3][2];
    int *p = arr+1;
    int *mp[2]={arr, tab[2]};
    int **pp = &p, **qq = mp;
    int (*s)[2] = tab;
}
```

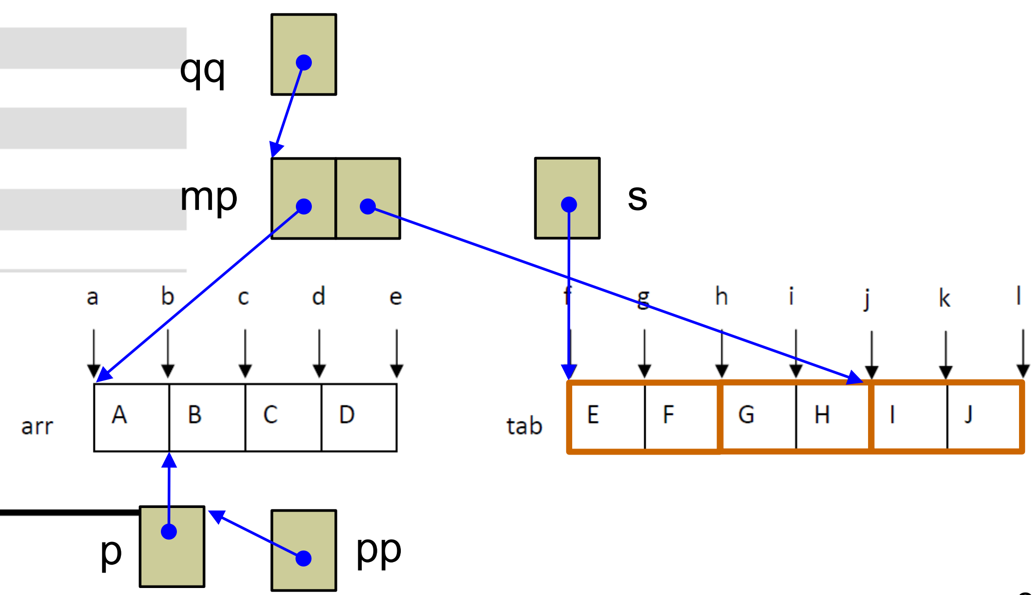


Expression
arr
arr[0]
arr+1
tab+1
tab[1]
tab[1]+1
tab[1][1]
&tab[1][1]
&tab[1][1]+1
p+0
p+1
*p
p[2]
mp[0][1]
mp[1][1]
*pp+2
(*pp)[1]
qq[1]
qq[1]+1
qq[0][0]
qq[1][0]
s[0][1]
s[1][0]

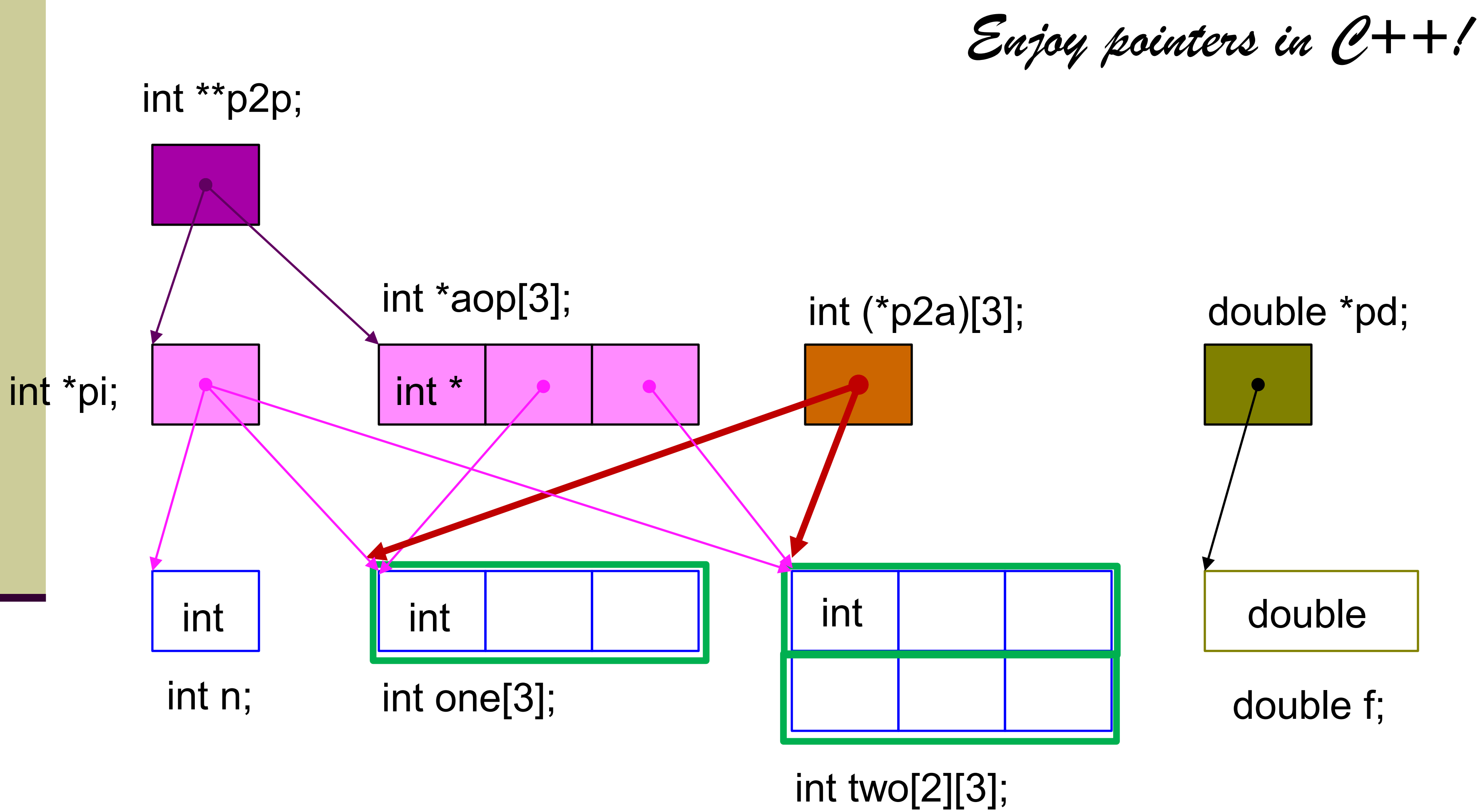
Exercise

Expression	Address or variable	Data type
arr	{A, B, C, D}	int [4]
arr[0]	A	int
arr+1	b	int *
tab+1		
tab[1]		
tab[1]+1		
tab[1][1]		
&tab[1][1]		
&tab[1][1]+1		
p+0		
p+1		
*p		
p[2]		
mp[0][1]		
mp[1][1]		
*pp+2		
(*pp)[1]		
qq[1]		
qq[1]+1		
qq[0][0]		
qq[1][0]		
s[0][1]		
s[1][0]		

```
int main()
{
    int arr[4];
    int tab[3][2];
    int *p = arr+1;
    int *mp[2]={arr, tab[2]};
    int **pp = &p, **qq = mp;
    int (*s)[2] = tab;
}
```



Illustration



Parameters vs. Non-parameters

```
void print1Darray1(int ptr1[], int size) {
   for (int i=0; i<size; i+=1) { printf("%d ", ptr1[i]); }
}
void print1Darray2 (int *ptr2, int size) {
   for (int i=0; i<size; i+=1) { printf("%d ", *(ptr2+i)); }
}
void print1Darray3 (int ptr3[10], int size) {
   for (int i=0; i<size; i+=1, ptr3+=1) { printf("%d ", *ptr3); }
}
void print1Darray4 (int ptr4[1000], int size) {
   for (int i=0; i<size; i+=1) { printf("%d ", ptr4[i]); }
}

int main()
{
   int arr1[],
       *arr2=0,
       arr3[10]={},
       arr4[1000]={};

   print1Darray1(arr3, 10);
   print1Darray2(arr4, 1000);
}
```

`ptr1`, `ptr2`, `ptr3`, and `ptr4` are all pointers to integer.

// syntax error, but `int arr1[] = {1, 2, 3};` defines an array of 3 integers.

// `arr2` is a pointer to integer

// `arr3` is an array of 10 integers

// `arr4` is an array of 1000 integers

In C++, we should initialize `arr2` by `nullptr`.
We will talk about `nullptr` later.

Parameters vs. Non-parameters

```
void print2Darray1(int ptr5[3][5], int rows) {
   for (int i=0; i<rows; i+=1) { print1Darray1(ptr5[i], 5); }
}
void print2Darray2(int (*ptr6)[5], int rows) {
   for (int i=0; i<rows; i+=1) {
      for (int j=0; j<5; j+=1) { printf("%d ", ptr6[i][j]); }
   }
}
void print2Darray3(int ptr7[][5], int rows) {
   for (int i=0; i<rows; i+=1) {
      for (int j=0; j<5; j+=1) { printf("%d ", *(*(ptr7+i)+j)); }
   }
}

int main()
{
   int arr5[3][5]={},
       arr6[4][5]={},
       arr7[4][4]={};

   print2Darray1(arr5, 3);
   print2Darray2(arr6, 4);
   print2Darray3(arr7, 4);
}
```

ptr5, ptr6, and ptr7 are all pointers to “array of 5 integers.”

// syntax error, incompatible type

Pointer to Struct/Class

```
struct Student
{
   int id, scores[3];
   char name[20];
};

void InputByUsers(Student *s)
{
   printf("Please input id and names...>");
   scanf("%d %s", &s->id, s->name );

   printf("Please input 3 scores...>");
   for (int i=0; i<3; i+=1)
       scanf("%d",  &s->scores[i] );
}

int main()
{
   Student stu;
   InputByUsers(&stu);
}
```

Note: Again, the coding style in the example is not good.
1. The magic number `3` appears three times.
2. Data members of `Student` are not default-initialized.

Pointer to Struct/Class

```
void InputByUsersBatch(Student stu[], int size)
{
   for (int s=0; s<size; s+=1)
   {
      printf("Student %d:\n", s+1);
      printf("Please input id and names...>");
      scanf("%d %s", &stu[s].id, stu[s].name);

      printf("Please input 3 scores...>");
      for (int i=0; i<3; i+=1)
         scanf("%d", &stu[s].scores[i] );
   }
}

int main()
{
    Student stu[10];
    InputByUsersBatch(stu, 10);
}
```

```
struct Student
{
   int id, scores[3];
   char name[20];
};
```

Pointer to Struct/Class

```
void InputByUsersBatch(Student stu[], int size)
{
   for (int s=0; s<size; s+=1)
   {
      printf("Student %d:\n", s+1);
      InputByUsers( &stu[s]  );
   }
}

int main()
{
    Student stu[10];
    InputByUsersBatch(stu, 10);
}
```

```
void InputByUsers(Student *s)
{
   ...
}
```

Qualifier `const`

- Passing the starting address of an array provides the full access.
- In most cases, we only need to read the array.

```c
#include <stdio.h>
void print(int p[], int size)
{
    for (int i=0; i<size; i+=1)
    {
       printf("%d ", p[i]);
    }

    p[0] = 0; // This should not happen.
}

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

   print(arr, 5);    // 11 22 33 44 55
   print(arr, 5);    // 0 22 33 44 55
}
```

Qualifier `const`

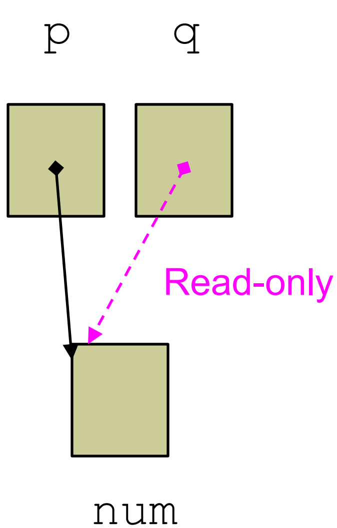
- With the full access, all functions are dangerous.

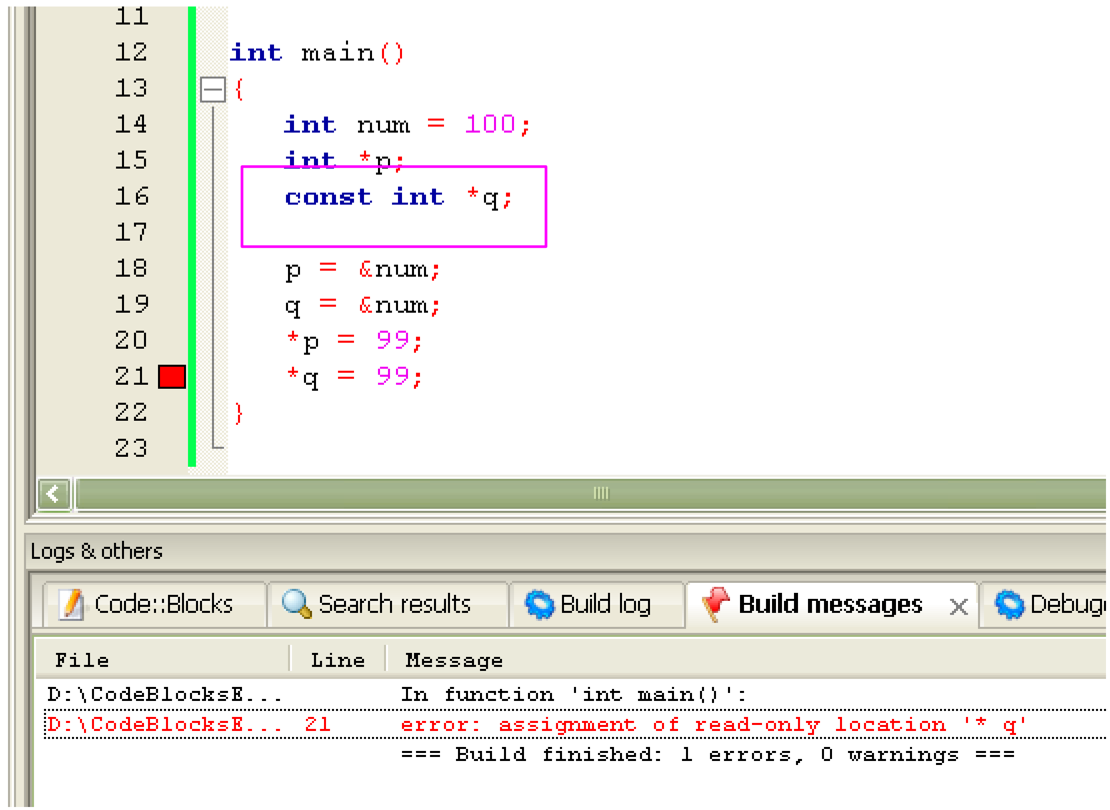
```
void print(int p[], int size)
{
    for (int i=0; i<size; i+=1)
    {
       printf("%d ", p[i]);
    }
}

// ... f1(), f2(), ..., f100()

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

   f1(arr, 5);
   f2(arr, 5);
   f3(arr, 5);

   f98(arr, 5);
   f99(arr, 5);
   f100(arr, 5);

   print(arr, 5); // 0 22 33 44 55
}
```

All functions `f1()`, `f2()`, …, `f100()` must be checked!

Qualifier `const`

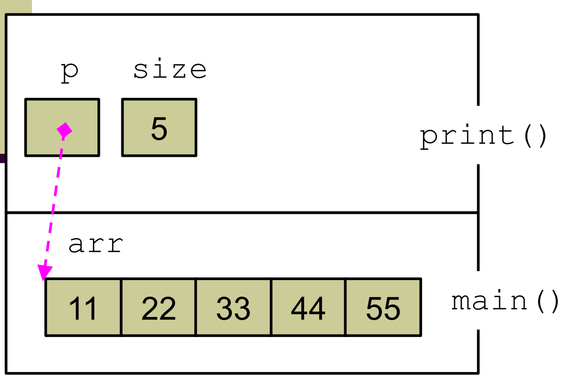
- Pointer-to-const
  - The pointer itself can be modified.
  - The pointed variable can NOT be modified through the pointer.

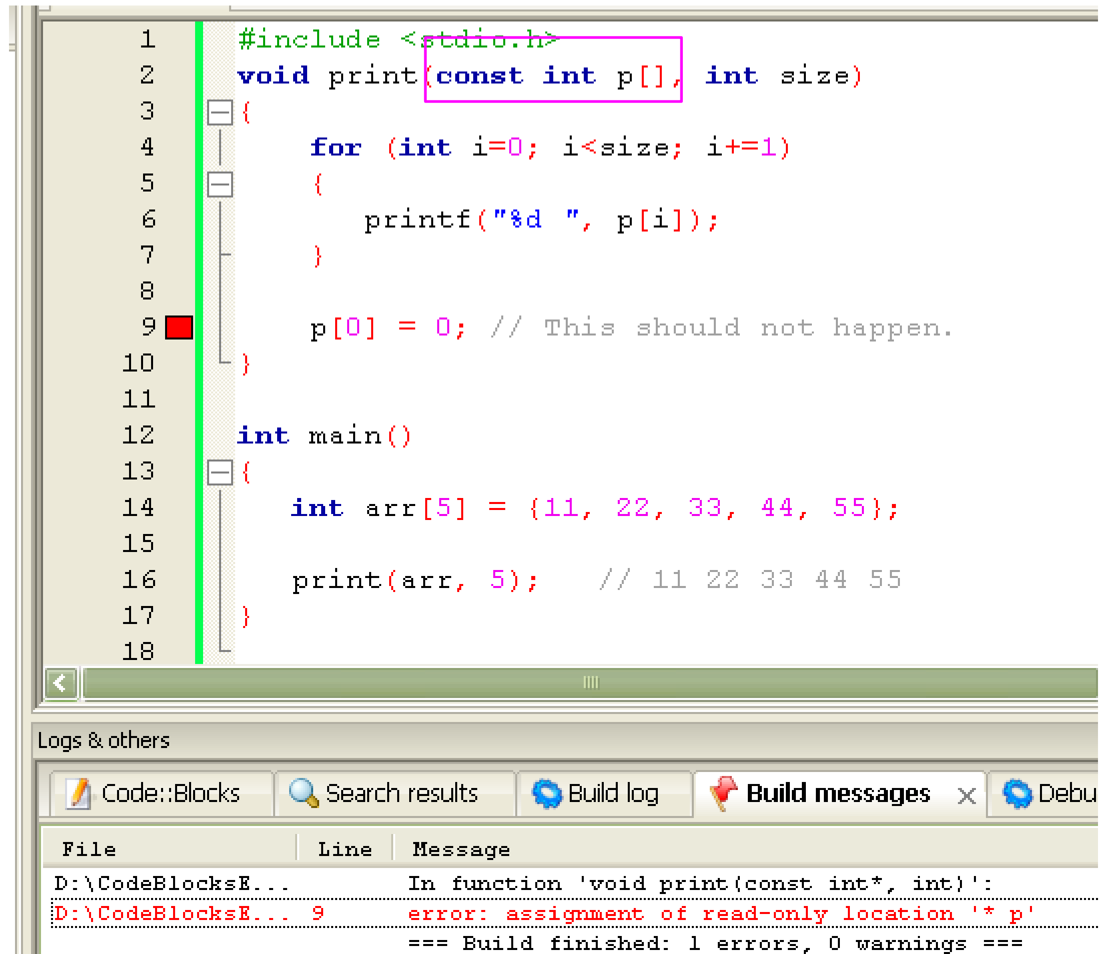




Qualifier `const`

- Pointer-to-const





Qualifier `const`

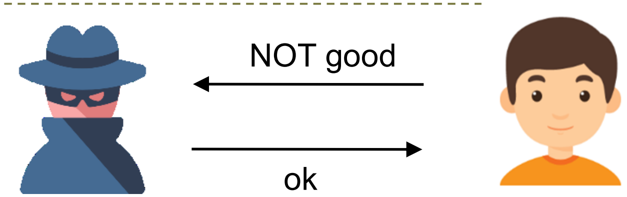
- Make the pointers point to const data whenever you can.

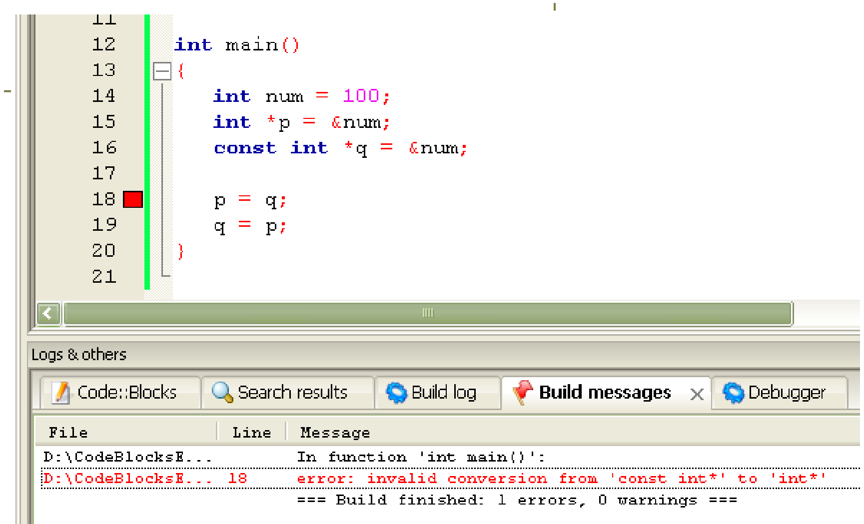
```
void print(const int p[], int size)
{
     for (int i=0; i<size; i+=1)
     {
        printf("%d ", p[i]);
     }
}

// ... f1(), f2(), ..., f100()

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

   f1(arr, 5);
   f2(arr, 5);
   f3(arr, 5);

   f98(arr, 5);
   f99(arr, 5);
   f100(arr, 5);

   print(arr, 5); // 0 22 33 44 55
}
```

I only have to check the functions whose parameter is a pointer-to-non-const.

Qualifier `const`

```
int main()
{
    int num = 100;
    int *p = &num;
    const int *q = &num;

    // This is not allowed.
    // Otherwise, the read-only guarantee is broken easily.
    p = q;

    // This is allowed.
    // Accessing num through q is safer than through p.
    q = p;
}
```





Icon made by Freepik from www.flaticon.com
Thief icons created by Chanut-is-Industries - Flaticon

Qualifier `const`

- The compiler ensures that you cannot call a function that may modify the pointed data inside a read-only function.

```c
2
3  void reset(int p[], int size)
4  {
5      for (int i=0; i<size; i+=1)
6      {
7          p[i] = 0;
8      }
9  }
10
11 void print(const int p[], int size)
12 {
13     for (int i=0; i<size; i+=1)
14     {
15         printf("%d ", p[i]);
16     }
17
18     reset(p, size);
19 }
20
21 int main()
22 {
23
24 }
```

Logs & others

Code::Blocks | Search results | Build log | Build messages | Debugger

File	Line	Message
D:\CodeBlocksE...		In function 'void print(const int*, int)':
D:\CodeBlocksE...	18	error: invalid conversion from 'const int*' to 'int*'
D:\CodeBlocksE...	18	error: initializing argument 1 of 'void reset(int*, int)'
		=== Build finished: 2 errors, 0 warnings ===

Qualifier `const`

- Good coding habits could save your time.
  - In this example, you might not check the code in `careless()` since it seems to do nothing to `b`.
  - If you always define a pointer-to-const in a read-only function, you can avoid such careless mistakes.

main.cpp

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

void careless(int arr[], int size)
{
    arr[size] = 0;
}

int main()
{
    int a[5] = {1, 2, 3, 4, 5}, b[5] = {3, 2, 7, 6, 8};
    printf("%p %p\n", a, b);
    printf("%d\n", b[0]);
    careless(a, 5);
    printf("%d\n", b[0]);
}
```

C:\Users\User\Desktop\main.exe

```
0061fee8 0061fefc
3
0

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

Qualifier const

- Note: adding const before and after the asterisk result in different data types.

```
int main()
{
   int num = 100;
   int *p;
   const int *q1; // q1 and q2 are the same type
   int const *q2; // They are pointer-to-const int

   q1 = &num;
//    *q1 = 99;     // The variable pointed by q1 cannot be modified through q1.

   q2 = &num;
//    *q2 = 99;

   int * const r = &num; // r is itself a constant.

//    r = &num;     // r cannot be modified.
   *r = 99;        // The variable pointed by r can be modified through r.
}
```

References

- Reference data type
  - References are like pointers, and using references is more convenient (but sometimes also confusing).
  - A reference is an alias (別名) of an existing object.
- To define a reference, use & in the definition.

```
#include <stdio.h>
int main()
{
    int num;
    int &ref = num; // ref is a reference to num

    num = 100;
    printf("num = %d, ref = %d.\n", num, ref);

    ref = 200;
    printf("num = %d, ref = %d.\n", num, ref);
}
```

```
num = 100, ref = 100.
num = 200, ref = 200.
```

References

- The ampersand (&) has three meanings in C++.
  - Bit-wise "AND" operator
  - Address-of operator
  - Definition of a reference

Recall that the asterisk (*) also has three meanings.

```
#include <stdio.h>
int main()
{
   int x = 8, y = 7, z = 1;
   z = x & y; // z <- 0 (1000 & 0111)

   int &r = x;

   if (&r == &x)
   {
      puts("r and x now refer to the same variable.");
   }
}
```

References

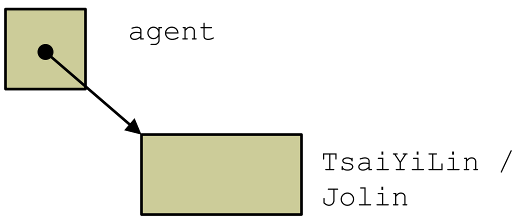
- Differences from pointers - 1
  - References must be initialized, and there is no null reference.

```
int main()
{
    int *p; // Pointers can be left uninitialized (not recommended).
    int *q = nullptr; // nullptr means 'q' points to nothing
    int &r; // error
}
```

`nullptr` is introduced in C++11 as a keyword to represent the null pointer.

References

- Differences from pointers - 2
  - Pointers are independent variables and have different types from what they point to.
  - References are just aliases.




```
#include <stdio.h>
int main()
{
    double TsaiYiLin;
    double *agent = &TsaiYiLin;
    double &Jolin = TsaiYiLin;

    // if (&TsaiYiLin != &agent); // error: incompatible data types
                                  // (double * vs. double **)

    if (sizeof(TsaiYiLin) == sizeof(Jolin)) { puts("The same size."); }
    if (&TsaiYiLin == &Jolin) { puts("The same variable."); }
}
```

```
The same size.
The same variable.
```

References

- Differences from pointers - 3
  - Pointers are independent variables and can be re-assigned.
  - References are just aliases and always refer to the same variable after being initialized.

```
#include <stdio.h>
int main()
{
    int f = 10, g = 20;
    int *p = &f; // p points to f
    int &r = f;  // r is f

    *p = 77; printf("f = %d, g = %d.\n", f, g);

    r = 88;  printf("f = %d, g = %d.\n", f, g);

    p = &g;
    *p = 99; printf("f = %d, g = %d.\n", f, g);

    r = g;   printf("f = %d, g = %d.\n", f, g);

    r = 100; printf("f = %d, g = %d.\n", f, g);
}
```

f = 77, g = 20.
f = 88, g = 20.
f = 88, g = **99**.
f = **99**, g = 99.
f = **100**, g = 99.

References

- Differences from pointers - 4
  - We can define an array of pointers, but we are not allowed to define an array of references.

```
int main()
{
    int *p[10];             // p is an array of 10 uninitialized pointers
    int *q[10] = {};         // q is an array of 10 null pointers
    int arr[10] = {};        // a is an array of 10 integers
    int &error[10];          // error: no array of references
    int (&ref)[10] = arr;    // ref refers to arr (ref is arr)
}
```

References

- The most common usage of references is to serve as parameters in functions.

```
#include <stdio.h>
void PassByValue(int v)
{
    printf("v = %d.\n", v);
    v = 0;
}
void PassByReference(int &r)
{
    printf("r = %d.\n", r);
    r = 0;
}
int main()
{
    int num = 10;

    printf("1: num = %d.\n", num);

    PassByValue(num);
    printf("2: num = %d.\n", num);

    PassByReference(num);
    printf("3: num = %d.\n", num);
}
```

```
1: num = 10.
v = 10.
2: num = 10.
r = 10.
3: num = 0.
```

References

- Without references, we pass the arguments to functions following these rules:
  - Pass the value of the argument of primitive types (e.g. `char`, `int`, `float`, `double`) or with small size by `T` if it is read-only in the function.
  - Pass the address of the argument of user-defined type (`struct/class`) or with large size (arrays) by `const T*` if it is read-only in the function.
  - Pass the address of the argument by `T*` if it is to be modified in the function.

References

- Passing arguments when we do not have references:

```c
void Print(int num)
{
    printf("%d", num);
}

void Set(int *p)
{
    *p = 100;
}

int main()
{
   int num;

   Set(&num);
   Print(num);
}
```

```c
struct Course
{
    int scores[1000] = {};
    int num_scores = 0;
};

void Print(const struct Course *c)
{
    for (int i=0; i<c->num_scores; i+=1)
    {
        printf("%3d", c->scores[i]);
    }
}

void Add(struct Course *c, int s)
{
    c->scores[c->num_scores] = s;
    c->num_scores += 1;
}

int main()
{
   struct Course course;
   Add(&course, 100);
   Print(&course); // a little bit strange
}
```

References

- With references, we pass the arguments to functions following these rules:
  - Pass the value of the argument of primitive types (e.g. `char`, `int`, `float`, `double`) or with small size by `T` if it is read-only in the function.
  - Pass arrays by `const T*` if it is read-only in the function.
  - Pass the argument of user-defined type (`struct`) by `const T&` if it is read-only in the function.
  - Pass the argument by `T*` `or` `T&` if it is to be modified in the function.
    - My habit is to pass by `T*`. There is no absolutely correct way.

References

- Passing arguments with references:

```
void Print(int num)
{
    printf("%d", num);
}

void Set(int *p)
{
    *p = 100;
}

int main()
{
   int num;

   Set(&num);
   Print(num);
}
```

```
struct Course
{
    int scores[1000] = {};
    int num_scores = 0;
};

void Print(const Course &c)
{
    for (int i=0; i<c.num_scores; i+=1)
    {
        printf("%d", c.scores[i]);
    }
}

void Add(Course *c, int s)
{
    c->scores[c->num_scores] = s;
    c->num_scores += 1;
}

int main()
{
   Course course;
   Add(&course, 100);
   Print(course); // looks better :p
}
```

References

- Which `swap` function(s) works correctly?

```
void swap1(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
void swap2(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
void swap3(int *a, int *b)
{
    int *tmp = a;
    a = b;
    b = tmp;
}
```

```
void swap4(int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
void swap5(int &a, int &b)
{
    int &tmp = a;
    a = b;
    b = tmp;
}
```

```
int main()
{
    int a = 0, b = 0;
    a = 3, b = 4; swap1(a, b); printf("%d %d\n", a, b);
    a = 3, b = 4; swap2(&a, &b); printf("%d %d\n", a, b);
    a = 3, b = 4; swap3(&a, &b); printf("%d %d\n", a, b);
    a = 3, b = 4; swap4(a, b); printf("%d %d\n", a, b);
    a = 3, b = 4; swap5(a, b); printf("%d %d\n", a, b);
}
```

```
3 4
4 3
3 4
4 3
4 4
```

References

- What's the output of the following program?

```
int f1(int v) { v = 1; return v; }
int f2(int &v) { v = 2; return v; }
int &f3(int v) { v = 3; return v; }    // we cannot return a local object by reference
int &f4(int &v) { v = 4; return v; }


int main()
{
    int a = 0, b = 0;
    b = f1(a); printf("%d %d\n", a, b);
    b = f2(a); printf("%d %d\n", a, b);
    b = f3(a); printf("%d %d\n", a, b);
    b = f4(a); printf("%d %d\n", a, b);
}
```

```
0 1
2 2
4 4
```

References

- What's the output of the following program?

```
int f1(int v) { v = 1; return v; }
int f2(int &v) { v = 2; return v; }
int & f4(int &v) { v = 4; return v; }

int main()
{
    int a1 = 0; int b1 = f1(a1); b1 += 1; printf("%d %d\n", a1, b1);
    int a2 = 0; int b2 = f2(a2); b2 += 1; printf("%d %d\n", a2, b2);
    int a3 = 0; int b3 = f4(a3); b3 += 1; printf("%d %d\n", a3, b3);
    int a4 = 0; int &b4 = f4(a4); b4 += 1; printf("%d %d\n", a4, b4);
}
```

```
0 2
2 3
4 5
5 5
```

Range `for` (C++11)

- Range `for` helps to do something to ALL elements in an array.

```
#include <stdio.h>
int main()
{
    int arr[5] = {8, 5, 3, 6, 2};

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

    printf("\n");
    for (int e: arr)
    {
        printf("%d ", e);
    }
}
```

Range `for`

- range `for` is not almighty.

```c
#include <stdio.h>
int main()
{
    int arr[50] = {8, 5, 3, 6, 2};

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

    for (int e: arr) // ?? 50 elements
    {
        printf("%d ", e);
    }
}
```

Sometimes I just want to process “some” elements.

```c
#include <stdio.h>
int main()
{
    int arr[5] = {8, 5, 3, 6, 2};

    for (int i=0; i<5; i+=1)
    {
        arr[i] += i;
    }

    for (int &e: arr)
    {
        // ?? no counter
    }
}
```

Sometimes I need the counter.

Range `for`

- Use references if you need to change the values.

```
int main()
{
    int arr[5] = {8, 5, 3, 6, 2};

    for (int e: arr)
    {
        e += 1;
    }

    for (int e: arr)
    {
        printf("%d ", e);
    }
}
```

8 5 3 6 2

```
int main()
{
    int arr[5] = {8, 5, 3, 6, 2};

    for (int &e: arr)
    {
        e += 1;
    }

    for (int e: arr)
    {
        printf("%d ", e);
    }
}
```

9 6 4 7 3

Range `for`

- A pointer is not an array (, again).

```
int main()
{
    int table[3][2] = {{2, 1}, {4, 3}, {6, 5}};

    for (int *p: table)
    {
        for (int i=0; i<2; i+=1) { printf("%d ", p[i]); }
//          for (int e: p) { printf("%d ", e); } // does not work

        printf("\n");
    }
}
```

```
int main()
{
    int table[3][2] = {{2, 1}, {4, 3}, {6, 5}};

    for (const int (&col)[2]: table)
    {
        for (int e: col) { printf("%d ", e); }
//          for (int i=0; i<2; i+=1) { printf("%d ", col[i]); } // also works

        printf("\n");
    }
}
```

C++11 keyword '`auto`' helps to make the code simpler.
`for (const auto &col: table)`