DASF004
Basic and Practice in Programming

Lecture 8

Pointer

Food for your MIND

Cyborg: A Design Project of a RCA Student

The Third Thumb
https://www.youtube.com/watch?v=pl5iTSA3L10

Agenda

Pointer Variable

- –Declaration and Initialization
- –Pointer Operations
- –Pointer and Array
- –Pointer as function arguments and return value

Review

Variable

- `int x=5;`
- `printf(“%d”,x); // value of a variable`

Address

- `int x=5;`
- `printf(“%d”,&x); // address of a variable`

Pass by reference

- Using address as argument of a function

C vs. C++

C Language does not allow reference operator &

C++ allows!

- For historical reason
- C++ (1979) was developed later than C (1969)
  - Most C++ implementation are back compatible with C
  - Some program mix C and C++ codes
- There are pros and cons for using reference operator

In Dev C++

- use .cpp file extension to indicate it is a C++ code

Global Variable

Global variables appear to be flexible
- It is in the global scope
- Flexible: easy to be abused

However, it is very bad for large scale code
- Can be modified by any scope
- Hard to manage and keep track of
- Making the program unpredictable, hard to debug

Imagine you found a bug in your program, and found that the global variable has a wrong value, how do you find out why the global variable is having a wrong value?

Avoid overusing of global variables
- One of the most common mishap for new programmers
- Don’t develop a bad habit at your early stage

Pointer Variable

Pointers

- Powerful, but difficult to master
- Complex pointer operations can be VERY complicated!
- Simulate call-by-reference
- Close relationship with arrays and character strings

Pointer Variable Declaration and Initialization

Pointer Variable

- Storing memory address location

Value	Address
0825	6087928
x: 7	6087932
a0b45	6087936
893f	6087940
Xptr: 6087932	6087944

- Normal variables contain a value

```
int x = 7;
```

- Pointer variables contain address of a variable

```
int * xPtr = &x;
```

The value of `x` is 7; the value of `xPtr` is an address (e.g. 6087932)

Pointer Variable Declaration

- Basic syntax: *Type *Name*
- Can declare pointers to any data type

Examples:

```
int *PPtr; /* P is var that can point to an int var */
float *QPtr;  /* Q is a float pointer */
char *RPtr;/* R is a char pointer */
```

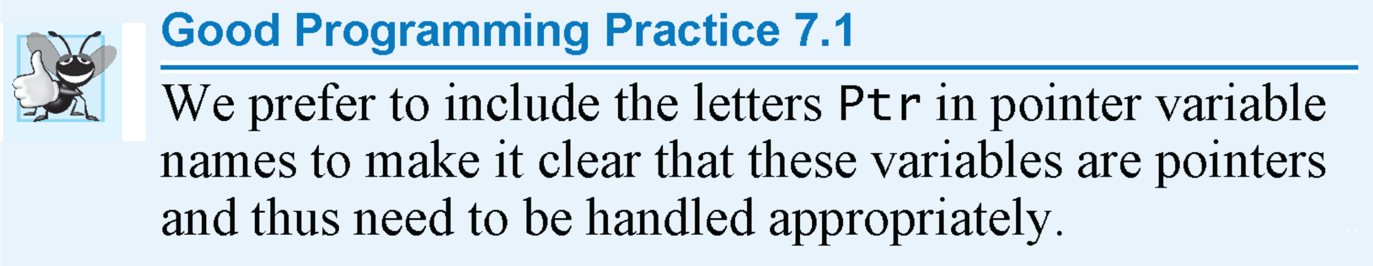
Complex example:

```
int *APtr[5]; /* AP is an array of 5 pointers to ints */
```

- Multiple pointers require using a **`*`** before each variable declaration

```
        int *myPtr1, *myPtr2;
```

More on how to read complex declarations later……



Good Programming Practice 7.1

We prefer to include the letters `Ptr` in pointer variable names to make it clear that these variables are pointers and thus need to be handled appropriately.

Pointer Variable Initialization

- Initialize pointer variable pointing to a variable

```
int x = 256;
int *xPtr = &x;  // *xPtr pointing to variable x
```

- Initialize pointers to **0**, or **NULL**
  - **0** or **NULL** – pointing to nothing (**NULL** preferred)

```
int *myPtr1 = 0;
int *myPtr1 = NULL;   // This is preferred
```

- **We never initialize pointer variables using an address**
  - `e.g. int * xPtr = 6087932;  // Error! We never do this!!!`

Pointer Operations

- `*xPtr` – the value of the item xPtr pointing to.
- Consider the following code segment:

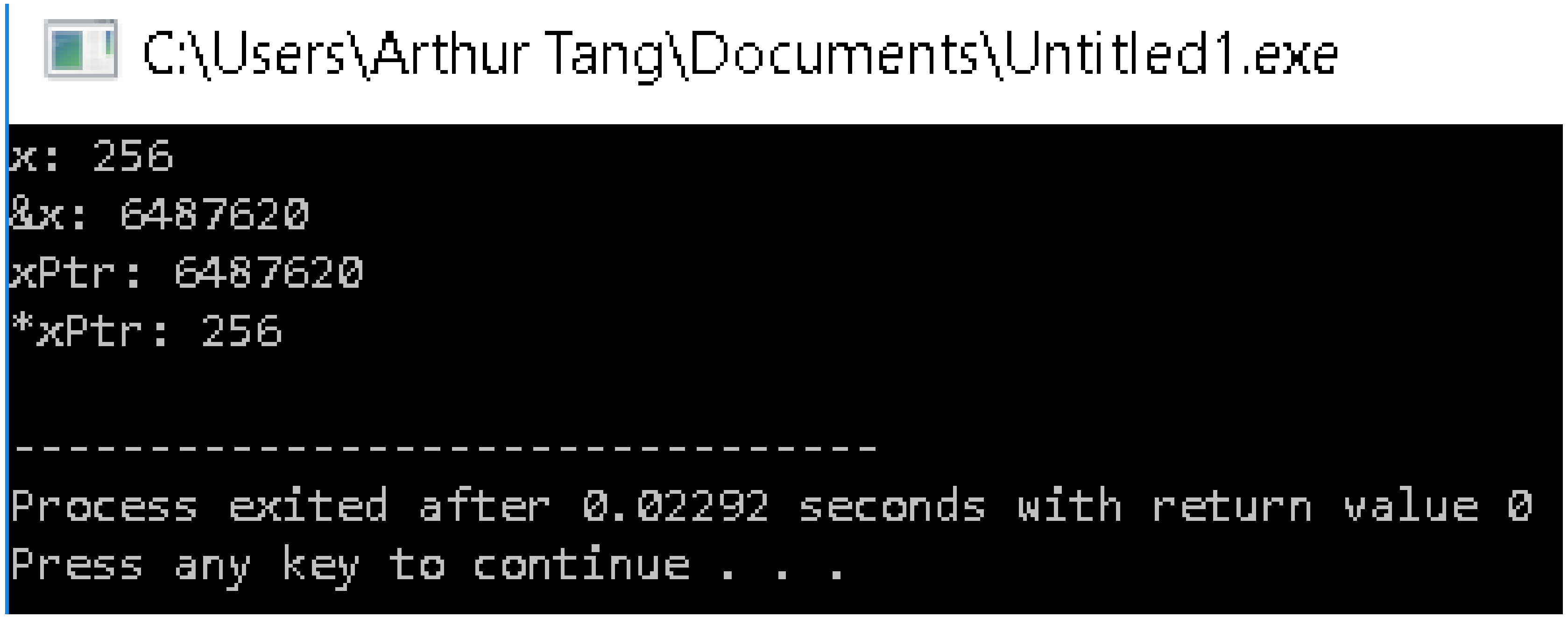
```
int x = 256;
int *xPtr = &x;         // *xPtr pointing to variable x
printf(“x: %d\n“,x);      // x: 256
printf(“&x: %d\n“,&x);  // &x: address of x
printf(“xPtr: %d\n“,xPtr)  // xPtr: address of x
     // Note: Value of xPtr is the address of x
printf(“*xPtr: %d\n“,*xPtr)  // *xPtr: 256
     // Note: Value of of the item xPtr pointing
```

x: value of x

&x: address of x

xPtr: value of xPtr; equals to the address of `x`

***xPtr:** value of the item xPtr pointing to

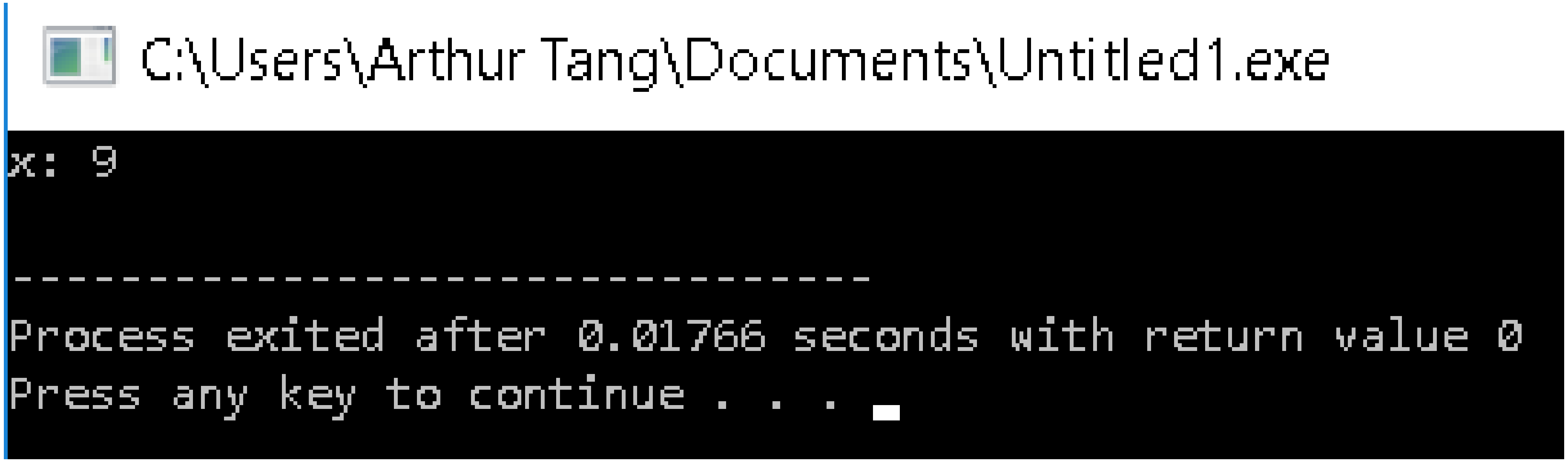


```
x: 256
&x: 6487620
xPtr: 6487620
*xPtr: 256

--------------------------------
Process exited after 0.02292 seconds with return value 0
Press any key to continue . . .
```

Pointer Operations

- Consider the following code segment:

```
int x = 256;
int *xPtr = &x;        // *xPtr pointing to variable x
*xPtr = 9;
printf(“x: %d”,x); // What is the value of x???
```



C:\Users\Arthur Tang\Documents\Untitled1.exe

```
x: 9


--------------------------------
Process exited after 0.01766 seconds with return value 0
Press any key to continue . . .
```

Pointer Operations

- Consider the following code segment:

```
int x = 5, y = 10;
int *Ptr = &x;
int **pPtr = &Ptr;
*Ptr = 3;
**pPtr = 7;
Ptr = &y;
**pPtr = 9;
*pPtr = &x;
*Ptr = -2;
```

738a389

8ff9f01

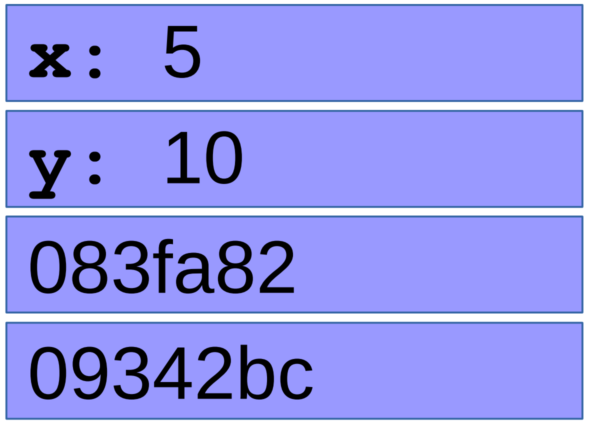
083fa82

09342bc

Pointer Operations

- Consider the following code segment:

```
int x = 5, y = 10;
int *Ptr = &x;
int **pPtr = &Ptr;
*Ptr = 3;
**pPtr = 7;
Ptr = &y;
**pPtr = 9;
*pPtr = &x;
*Ptr = -2;
```



x: 5

y: 10

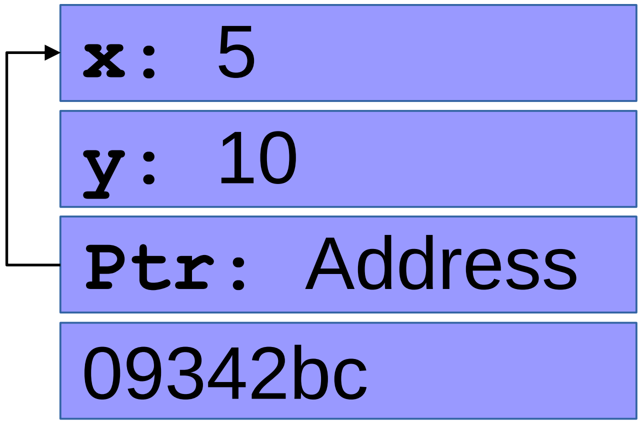
083fa82

09342bc

Pointer Operations

- Consider the following code segment:

```
int x = 5, y = 10;
int *Ptr = &x;
int **pPtr = &Ptr;
*Ptr = 3;
**pPtr = 7;
Ptr = &y;
**pPtr = 9;
*pPtr = &x;
*Ptr = -2;
```



x: 5

y: 10

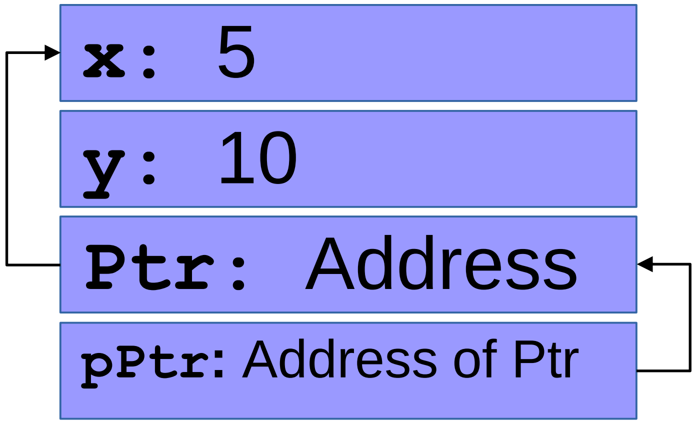
Ptr: Address

09342bc

Pointer Operations

- Consider the following code segment:

```
int x = 5, y = 10;
int *Ptr = &x;
int **pPtr = &Ptr;
*Ptr = 3;
**pPtr = 7;
Ptr = &y;
**pPtr = 9;
*pPtr = &x;
*Ptr = -2;
```

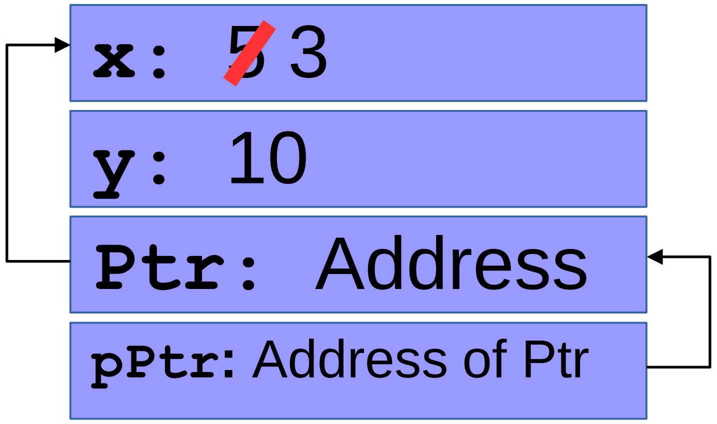


A Pointer of a Pointer, pointing to the address of a Pointer

Pointer Operations

- Consider the following code segment:

```
int x = 5, y = 10;
int *Ptr = &x;
int **pPtr = &Ptr;
*Ptr = 3;
**pPtr = 7;
Ptr = &y;
**pPtr = 9;
*pPtr = &x;
*Ptr = -2;
```


x: ~~5~~ 3

y: 10

Ptr: Address

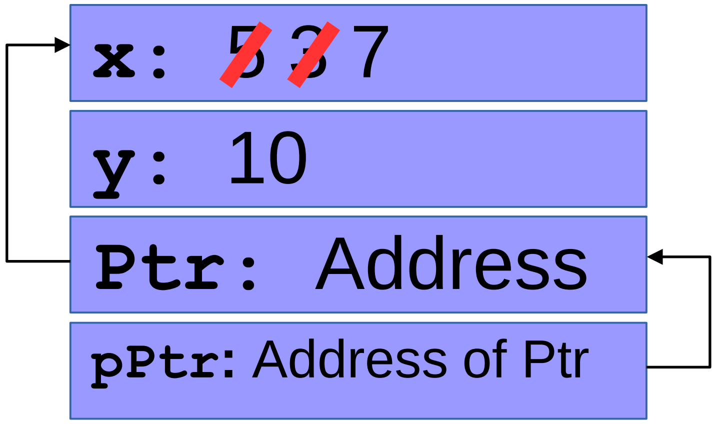
pPtr: Address of Ptr

Assign the value 3 to the item Ptr pointing to

Pointer Operations

- Consider the following code segment:

```
int x = 5, y = 10;
int *Ptr = &x;
int **pPtr = &Ptr;
*Ptr = 3;
**pPtr = 7;
Ptr = &y;
**pPtr = 9;
*pPtr = &x;
*Ptr = -2;
```

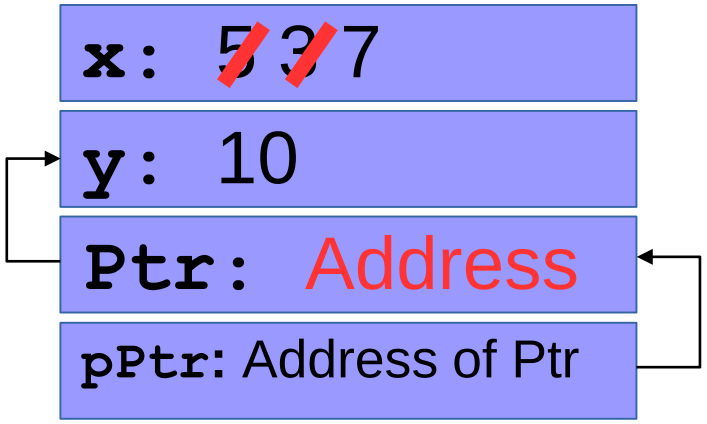


Assign the value 7 to item of the item pPtr pointing to

Pointer Operations

- Consider the following code segment:

```
int x = 5, y = 10;
int *Ptr = &x;
int **pPtr = &Ptr;
*Ptr = 3;
**pPtr = 7;
Ptr = &y;
**pPtr = 9;
*pPtr = &x;
*Ptr = -2;
```



x: ~~5~~ ~~3~~ 7

y: 10

Ptr: Address

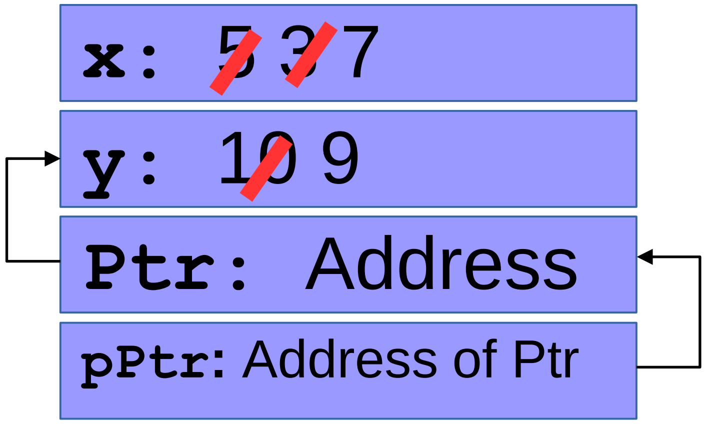
pPtr: Address of Ptr

Ptr now pointing to the address of y

Pointer Operations

- Consider the following code segment:

```
int x = 5, y = 10;
int *Ptr = &x;
int **pPtr = &Ptr;
*Ptr = 3;
**pPtr = 7;
Ptr = &y;
**pPtr = 9;
*pPtr = &x;
*Ptr = -2;
```

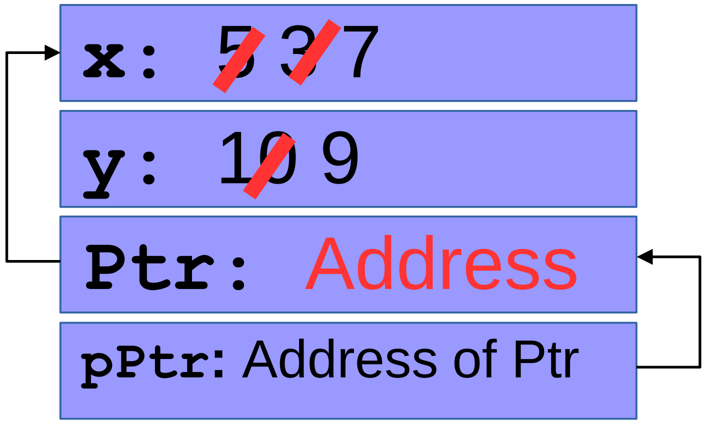


Assign the value 9 to item of the item pPtr pointing to

Pointer Operations

- Consider the following code segment:

```
int x = 5, y = 10;
int *Ptr = &x;
int **pPtr = &Ptr;
*Ptr = 3;
**pPtr = 7;
Ptr = &y;
**pPtr = 9;
*pPtr = &x;
*Ptr = -2;
```



x: ~~5~~ ~~3~~ 7

y: ~~10~~ 9

Ptr: Address

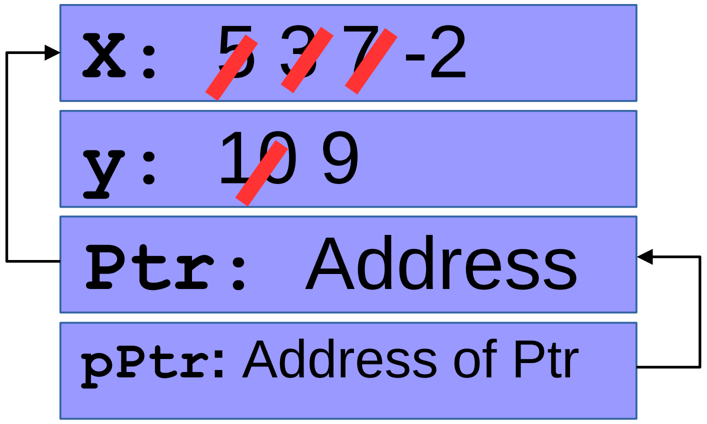
pPtr: Address of Ptr

Assign the address of x to the item pPtr pointing to

Pointer Operations

- Consider the following code segment:

```
int x = 5, y = 10;
int *Ptr = &x;
int **pPtr = &Ptr;
*Ptr = 3;
**pPtr = 7;
Ptr = &y;
**pPtr = 9;
*pPtr = &x;
*Ptr = -2;
```



x: ~~5~~ ~~3~~ ~~7~~ -2

y: ~~10~~ 9

Ptr: Address

pPtr: Address of Ptr

Assign the value -2 to the item Ptr pointing to

Pointer Arithmetic

What’s `ptr + 1`?

➔The next memory location!

What’s `ptr - 1`?

➔The previous memory location!

What’s `ptr * 2` and `ptr / 2`?

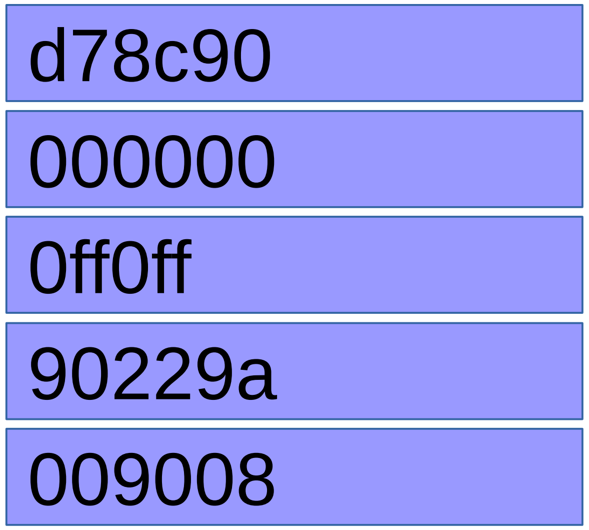
➔Invalid operations!!!

Pointer Arithmetic operations should be performed on array, otherwise it is meaningless!!!

Pointer Operations and Array

- Consider the following code segment:

```
float a[4];
float *aPtr = NULL;
aPtr = &(a[2]);
*aPtr = 3.14;
aPtr++;
*aPtr = 9.0;
aPtr = aPtr - 3;
*aPtr = 6.0;
aPtr += 2;
*aPtr = 7.0;
*(aPtr-1) = 12.5;
```



d78c90

000000

0ff0ff

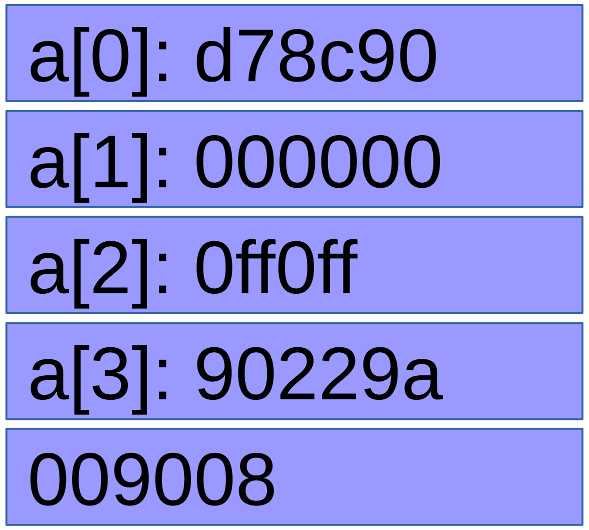
90229a

009008

Pointer Operations and Array

- Consider the following code segment:

```
float a[4];
float *aPtr = NULL;
aPtr = &(a[2]);
*aPtr = 3.14;
aPtr++;
*aPtr = 9.0;
aPtr = aPtr - 3;
*aPtr = 6.0;
aPtr += 2;
*aPtr = 7.0;
*(aPtr-1) = 12.5;
```



a[0]: d78c90

a[1]: 000000

a[2]: 0ff0ff

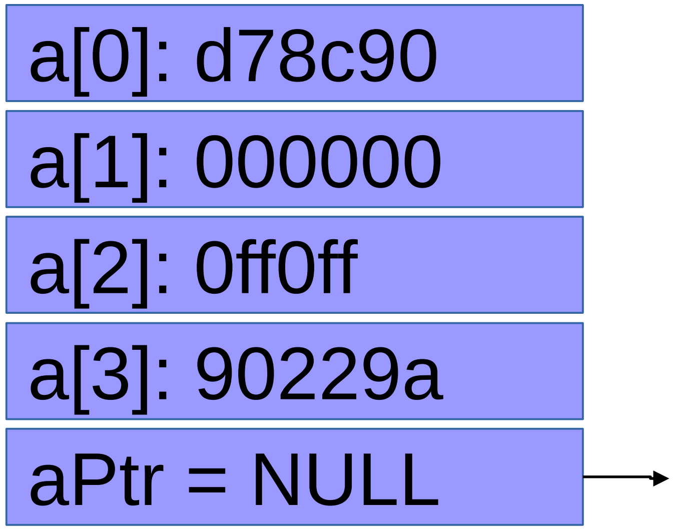
a[3]: 90229a

009008

Pointer Operations and Array

- Consider the following code segment:

```
float a[4];
float *aPtr = NULL;
aPtr = &(a[2]);
*aPtr = 3.14;
aPtr++;
*aPtr = 9.0;
aPtr = aPtr - 3;
*aPtr = 6.0;
aPtr += 2;
*aPtr = 7.0;
*(aPtr-1) = 12.5;
```



a[0]: d78c90

a[1]: 000000

a[2]: 0ff0ff

a[3]: 90229a

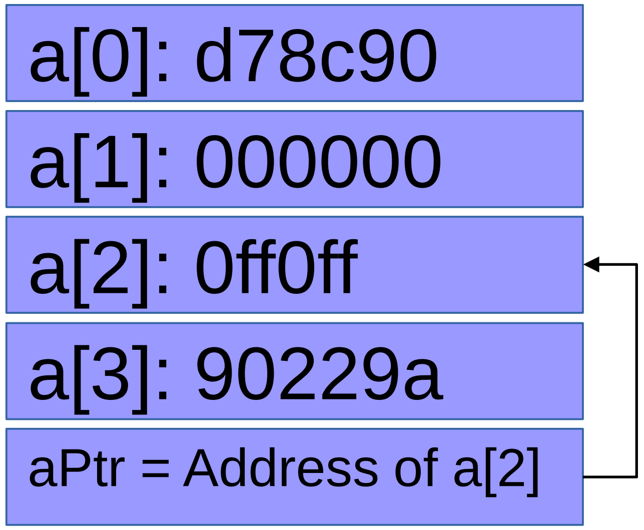
aPtr = NULL

An empty pointer
pointing to nothing

Pointer Operations and Array

- Consider the following code segment:

```
float a[4];
float *aPtr = NULL;
aPtr = &(a[2]);
*aPtr = 3.14;
aPtr++;
*aPtr = 9.0;
aPtr = aPtr - 3;
*aPtr = 6.0;
aPtr += 2;
*aPtr = 7.0;
*(aPtr-1) = 12.5;
```



a[0]: d78c90

a[1]: 000000

a[2]: 0ff0ff

a[3]: 90229a

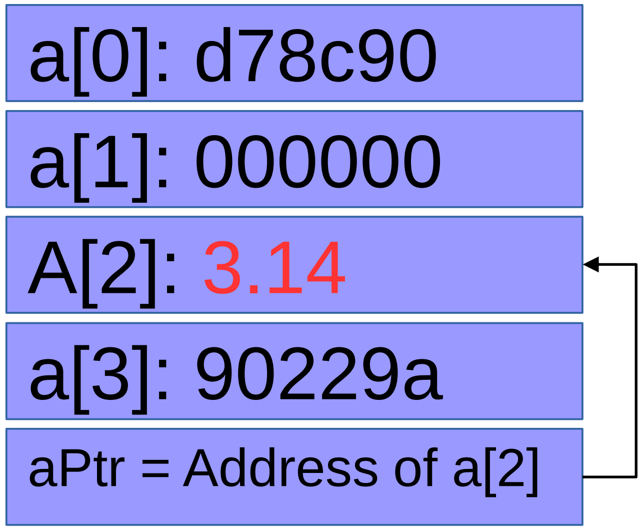
aPtr = Address of a[2]

Assign the address of a[2] to aPtr

Pointer Operations and Array

- Consider the following code segment:

```
float a[4];
float *aPtr = NULL;
aPtr = &(a[2]);
*aPtr = 3.14;
aPtr++;
*aPtr = 9.0;
aPtr = aPtr - 3;
*aPtr = 6.0;
aPtr += 2;
*aPtr = 7.0;
*(aPtr-1) = 12.5;
```

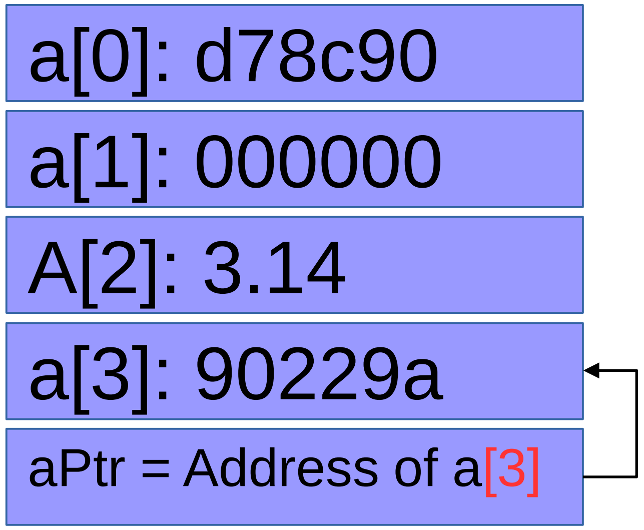


Assign value 3.14 to
item aPtr pointing to

Pointer Operations and Array

- Consider the following code segment:

```
float a[4];
float *aPtr = NULL;
aPtr = &(a[2]);
*aPtr = 3.14;
aPtr++;
*aPtr = 9.0;
aPtr = aPtr - 3;
*aPtr = 6.0;
aPtr += 2;
*aPtr = 7.00;
*(aPtr-1) = 12.5;
```



a[0]: d78c90

a[1]: 000000

A[2]: 3.14

a[3]: 90229a

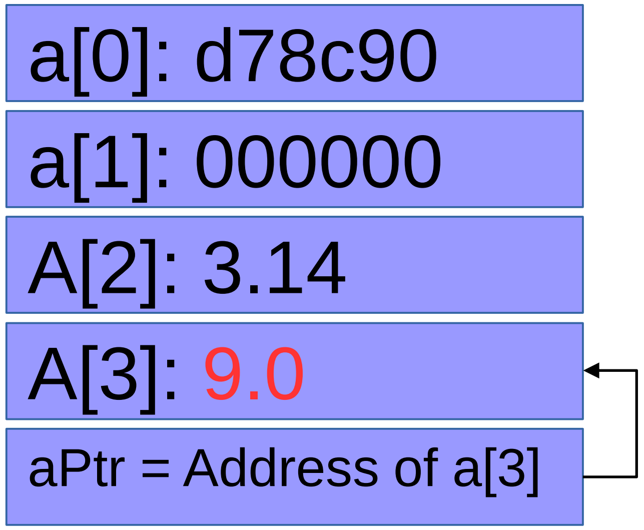
aPtr = Address of a[3]

Increase the address of aPtr by one float (since aPtr is a float pointer)

Pointer Operations and Array

- Consider the following code segment:

```
float a[4];
float *aPtr = NULL;
aPtr = &(a[2]);
*aPtr = 3.14;
aPtr++;
*aPtr = 9.0;
aPtr = aPtr - 3;
*aPtr = 6.0;
aPtr += 2;
*aPtr = 7.0;
*(aPtr-1) = 12.5;
```


a[0]: d78c90

a[1]: 000000

A[2]: 3.14

A[3]: 9.0

aPtr = Address of a[3]

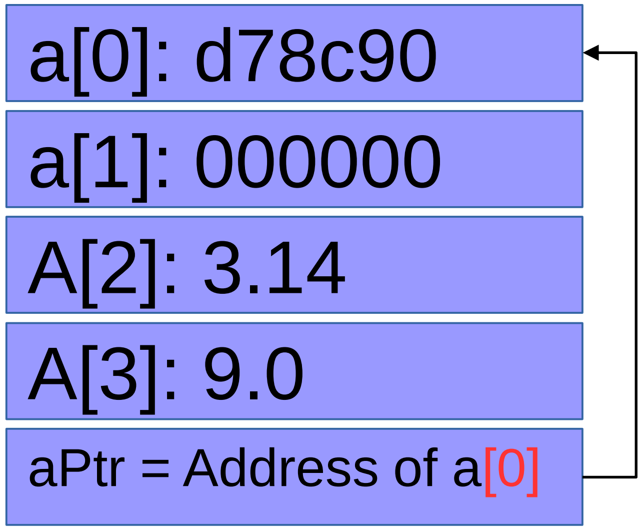
Assign value 9.0 to
item aPtr pointing to

Pointer Operations and Array

- Consider the following code segment:

```
float a[4];
float *aPtr = NULL;
aPtr = &(a[2]);
*aPtr = 3.14;
aPtr++;
*aPtr = 9.0;
aPtr = aPtr - 3;
*aPtr = 6.0;
aPtr += 2;
*aPtr = 7.0;
*(aPtr-1) = 12.5;
```

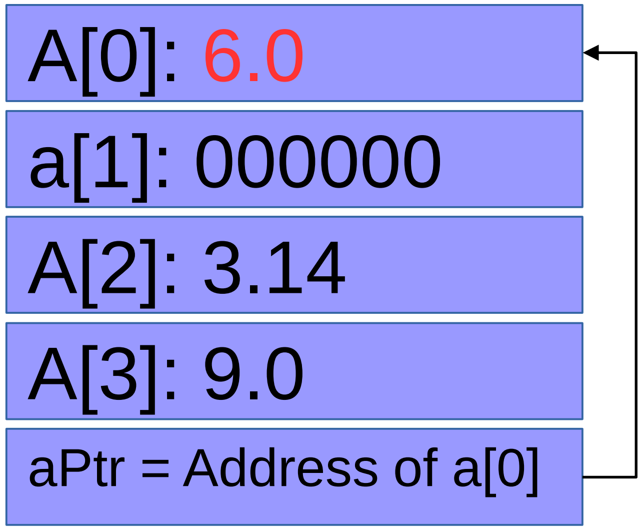
Decrease the address of aPtr by three float



Pointer Operations and Array

- Consider the following code segment:

```
float a[4];
float *aPtr = NULL;
aPtr = &(a[2]);
*aPtr = 3.14;
aPtr++;
*aPtr = 9.0;
aPtr = aPtr - 3;
*aPtr = 6.0;
aPtr += 2;
*aPtr = 7.0;
*(aPtr-1) = 12.5;
```



A[0]: 6.0

a[1]: 000000

A[2]: 3.14

A[3]: 9.0

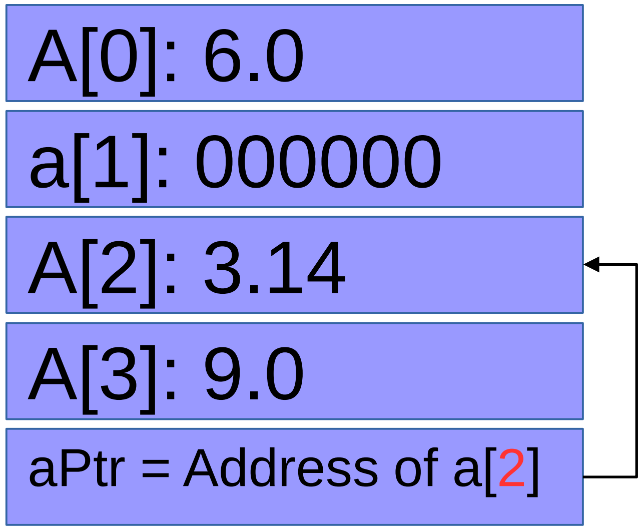
aPtr = Address of a[0]

Assign value 6.0 to
item aPtr pointing to

Pointer Operations and Array

- Consider the following code segment:

```
float a[4];
float *aPtr = NULL;
aPtr = &(a[2]);
*aPtr = 3.14;
aPtr++;
*aPtr = 9.0;
aPtr = aPtr - 3;
*aPtr = 6.0;
aPtr += 2;
*aPtr = 7.0;
*(aPtr-1) = 12.5;
```



A[0]: 6.0

a[1]: 000000

A[2]: 3.14

A[3]: 9.0

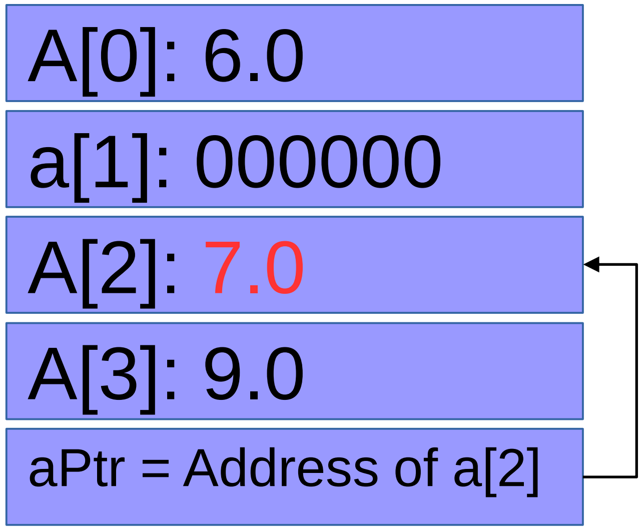
aPtr = Address of a[2]

Increase the address aPtr by two float

Pointer Operations and Array

- Consider the following code segment:

```
float a[4];
float *aPtr = NULL;
aPtr = &(a[2]);
*aPtr = 3.14;
aPtr++;
*aPtr = 9.0;
aPtr = aPtr - 3;
*aPtr = 6.0;
aPtr += 2;
*aPtr = 7.0;
*(aPtr-1) = 12.5;
```


A[0]: 6.0

a[1]: 000000

A[2]: 7.0

A[3]: 9.0

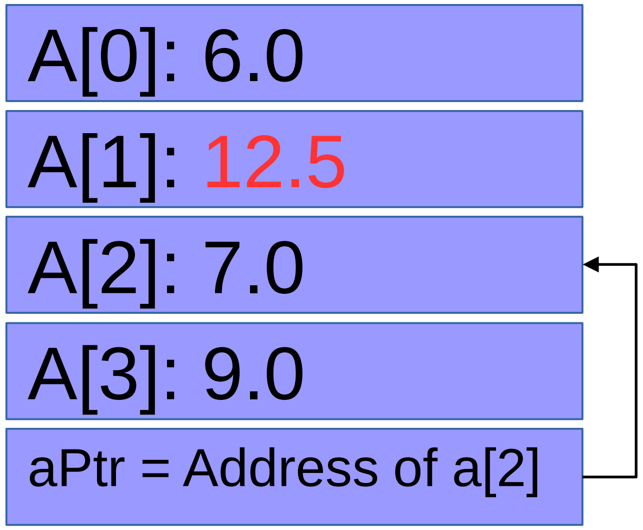
aPtr = Address of a[2]

```
Assign value 7.0 to
   item aPtr pointing to
```

Pointer Operations and Array

- Consider the following code segment:

```
float a[4];
float *aPtr = NULL;
aPtr = &(a[2]);
*aPtr = 3.14;
aPtr++;
*aPtr = 9.0;
aPtr = aPtr - 3;
*aPtr = 6.0;
aPtr += 2;
*aPtr = 7.0;
*(aPtr-1) = 12.5;
```



A[0]: 6.0

A[1]: 12.5

A[2]: 7.0

A[3]: 9.0

aPtr = Address of a[2]

Assign value 12.5 to the item previous to aPtr pointing to

Exercise

- Consider the following code segment:

```
int a[10];
int *aPtr = NULL;
aPtr = a;
for(int i=0;i<10;i++)
{ *aPtr = i;
  aPtr++;
}
printf("%d",a[5]);
```

What is the output of the code segment?

Pointer Operations

- Subtracting Pointers:

```
double a[4];
double *aPtr = &a[1];
double *bPtr = &a[3];
printf(“value: %d\n”,bPtr – aPtr);
Value of bPtr – aPtr is 2 (two double)
```

- Pointers Comparison:

```
aPtr < bPtr is TRUE
bPtr == aPtr is FALSE
```

Pointer Types

Pointers are generally of the same size (enough bytes to represent all possible memory addresses).

- `int *, float *, double *, char *` have the same size.

But it is inappropriate to assign an address of one type of variable to a different type of pointer (inappropriate, but valid and may result in logical error)

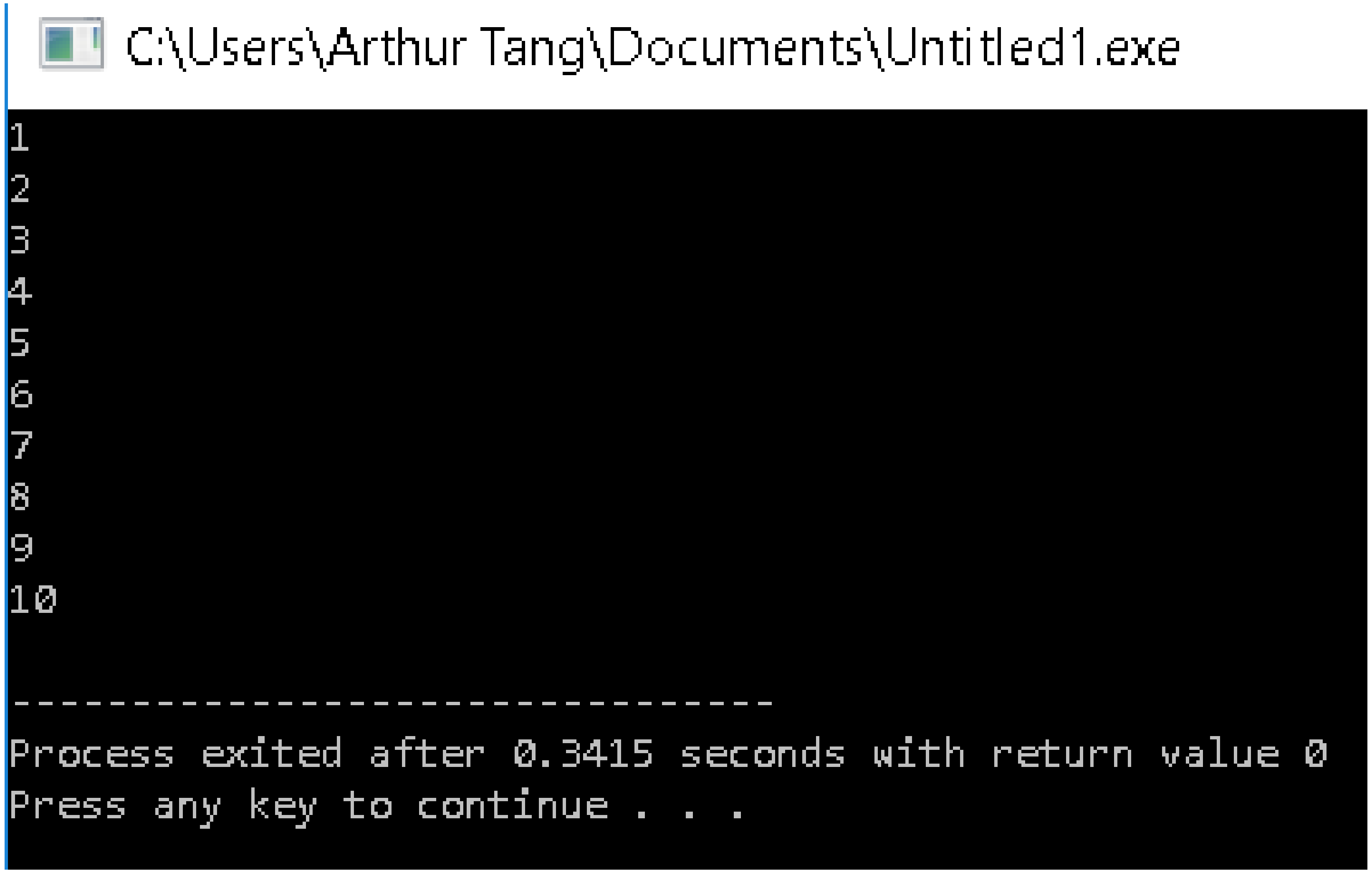
Example:

```
int V = 101;
float *P = &V; /* Generally results in a Warning */
```

Warning rather than error because C will allow you to do this (it is appropriate in certain situations)

Iterating through the array

```
int x[10] = {1,2,3,4,5,6,7,8,9,10};
int * xPtr = x;
for (int i=0;i<10;i++)
{ printf(“%d\n”,*xPtr);
  xPtr++;
}
```

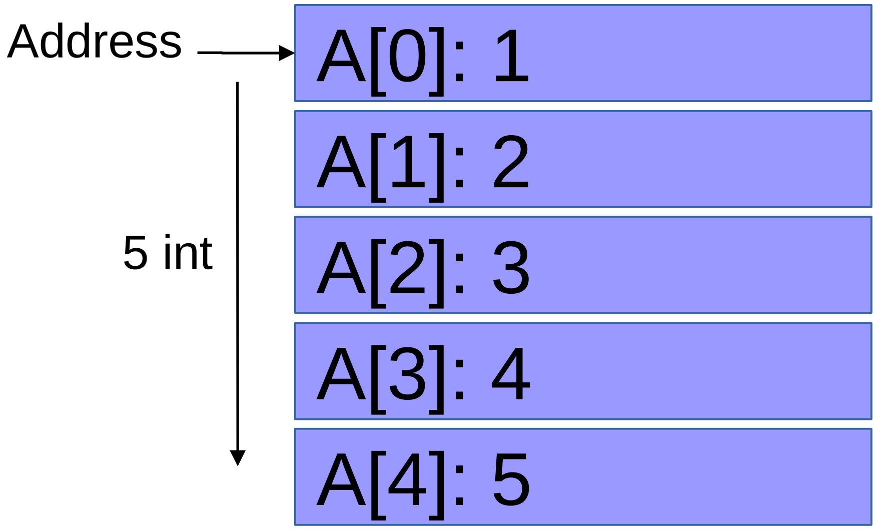


C:\Users\Arthur Tang\Documents\Untitled1.exe

```
1
2
3
4
5
6
7
8
9
10

--------------------------------
Process exited after 0.3415 seconds with return value 0
Press any key to continue . . .
```

Function with Pointer(s) As Argument(s)

```
#include <stdio.h>

int ArrayTotal(int * A, int size)
{ int total = 0;
  for(int i=0;i<size;i++)
    total += A[i];
  return total;
}

int main(void)
{ int A[5] = {1,2,3,4,5};
  int sum = ArrayTotal(A,5);
  printf(“Sum: %d”,sum);
  return 0;
}
```



Address

5 int

A[0]: 1

A[1]: 2

A[2]: 3

A[3]: 4

A[4]: 5

Function with Pointer(s) As Argument(s)

```
#include <stdio.h>

int ArrayTotal(int A[], int size)  // Same!
{ int total = 0;
  for(int i=0;i<size;i++)
    total += A[i];
  return total;
}

int main(void)
{ int A[5] = {1,2,3,4,5};
  int sum = ArrayTotal(A,5);
  printf(“Sum: %d”,sum);
  return 0;
}
```

Function with Pointer(s) As Argument(s)

When passing pointer(s) as functions’ arguments, it is passing by reference

i.e. If the functions modify the values of the items pointed by the pointers, the modification will be reflected in the function calling it

```
#include <stdio.h>

void ArrayFunction(int *A, int size)
{ int total = 0;
  for(int i=0;i<size;i++)
    A[i]++;               // Add one to each array item
}

int main(void)
{ int A[5] = {1,2,3,4,5};
  ArrayFunction(A,5);
  printf(“{%d,%d,%d,%d,%d}”,A[0],A[1],A[2],A[3],A[4]);
  return 0;
}
```

Function with Pointer As Return Value

Functions can return a pointer as return value

```
int * MyFunction(int x)
{ return &x;
}

int main(void)
{ int x = 123;
  int * xPtr = NULL;
  xPtr = MyFunction(x);
  printf(“%d”,*xPtr);
  return 0;
}
```

Pointer Return Values

```
float *findMax(float A[], int N) {
  int I;
  float *theMax = &(A[0]);

  for (I = 1; I < N; I++)
    if (A[I] > *theMax) theMax = &(A[I]);

  return theMax;
}

void main() {
  float A[5] = {0.0, 3.0, 1.5, 2.0, 4.1};
  float *maxA = NULL;

  maxA = findMax(A,5);
  *maxA = *maxA + 1.0;
  printf("%f %f\n",*maxA,A[4]);
}
```

Q&A?