Week 4 Lecture 1

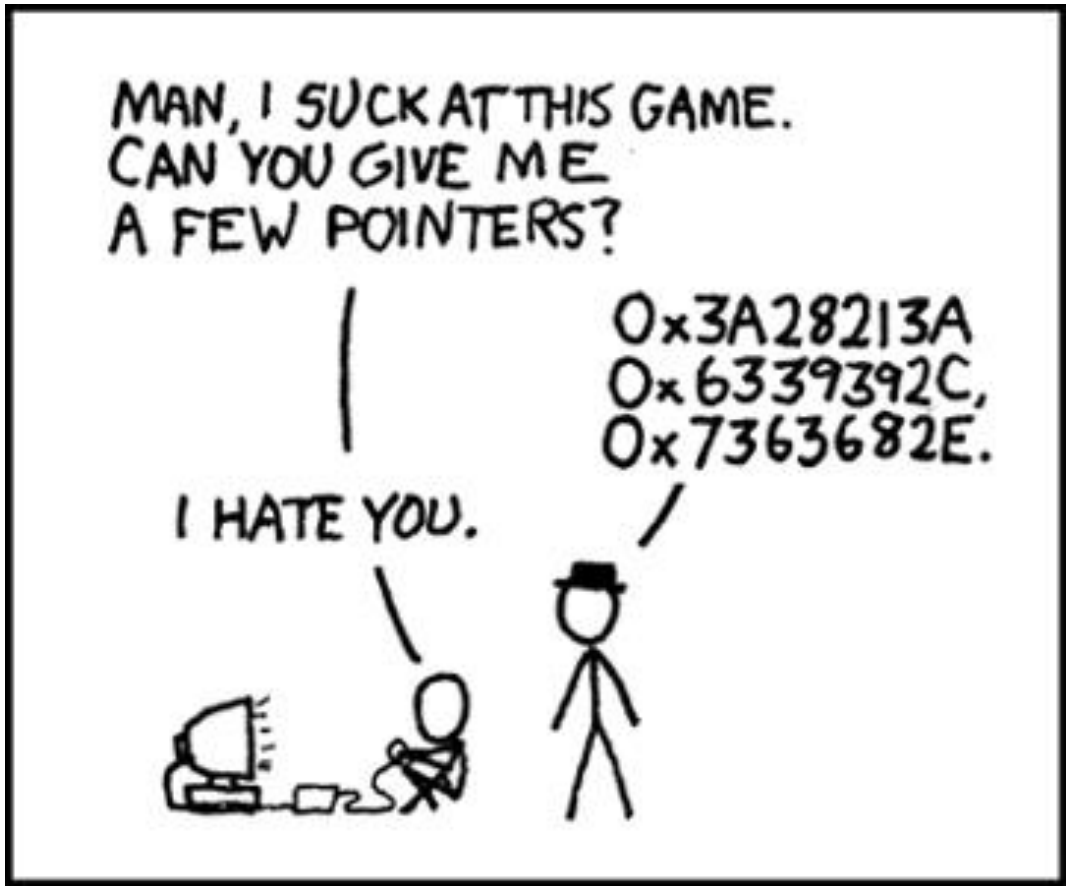
NWEN 241
Systems Programming

Jyoti Sahni

Jyoti.sahni@ecs.vuw.ac.nz

Content

- More on Pointers



Recap: Declaring a Pointer

- Pointers are typed based on the type of entity that they point to
  - To declare a pointer, use * preceding the variable name as in:

```
data_type *name;
```

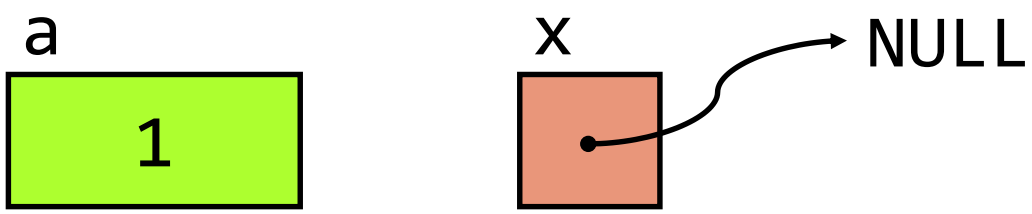
  - Examples:

```
int *p;    // p is a pointer to an int
float *q;  // q is a float pointer
char *r;   // r is a char pointer
int *s[5]; // s is an array of 5 int pointers
```

Recap: Graphical Illustration

Declaration:

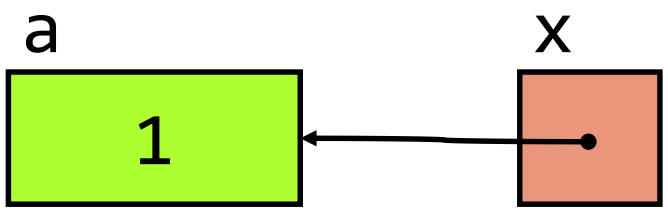
```
int a = 1; int *x = NULL;
```



NULL – pointer literal/constant to non-existent address

Assignment:

```
x = &a;
```



Recap: Usage of Pointers

1) Provide an alternative means of accessing information stored in arrays

2) Provide an alternative (and more efficient) means of passing parameters to functions

3) Enable dynamic data structures, that are built up from blocks of memory allocated from the heap at run time

Pointers and Arrays (1)

- **Arrays in C are pointed to**, i.e. the variable that you declare for the array is actually a **fixed pointer** to the first array element

- Example:

```
int z[10] = {1, 2, 3};
```

  - `z` is a fixed pointer, it points to the address of the first element `z[0]`
  - In other words, `z == &z[0]`

Pointers and Arrays (2)

- Array elements are usually accessed using [ ] (with the index)

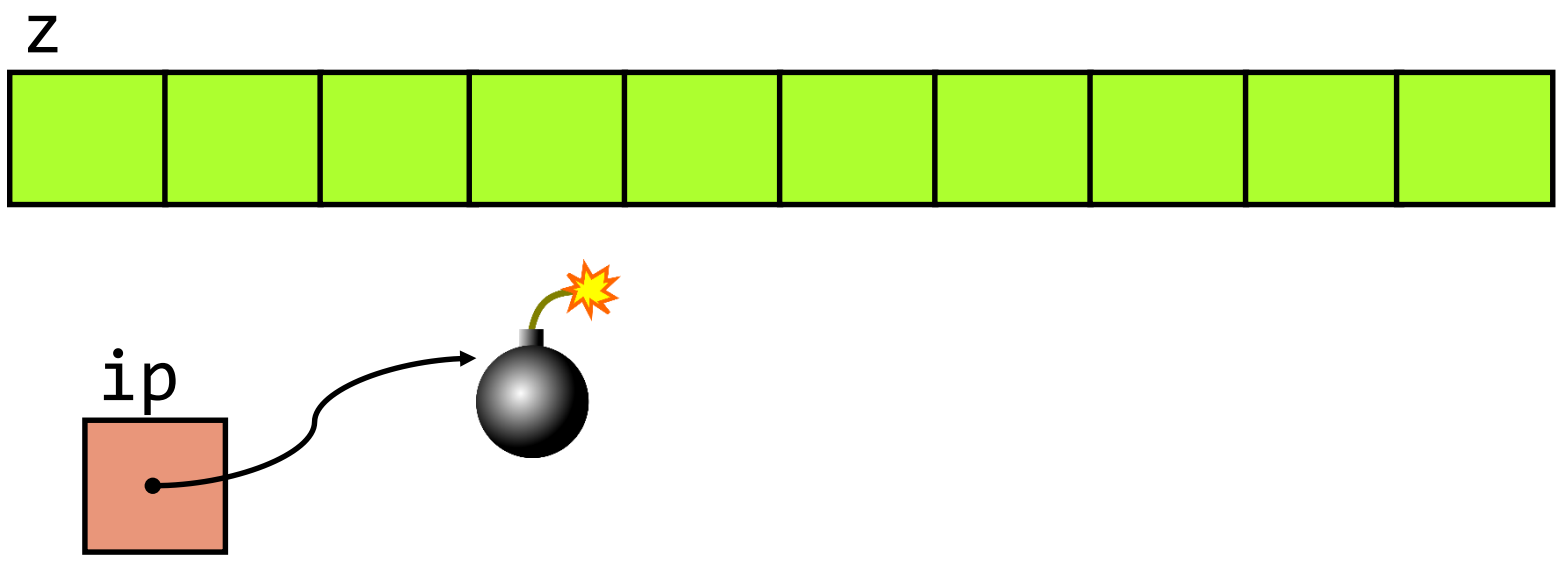
- Pointers can also be used to access array elements

```
int z[10], *ip;
ip = &z[0];
```

  - `z[0]`, `ip[0]`, `*z`, or `*ip` can all be used to access the *first* element of the array `z`

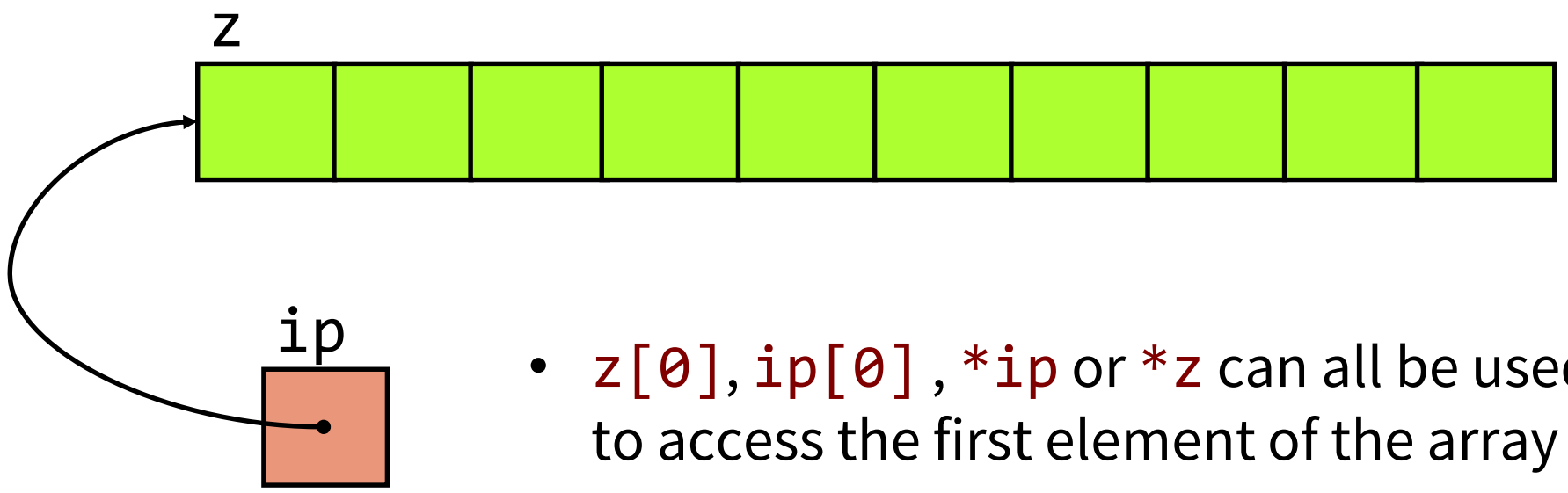
Graphical Illustration

```
int z[10], *ip;
ip = &z[0];
```



Graphical Illustration

```
int z[10], *ip;
ip = &z[0];
```



- `z[0]`, `ip[0]` , `*ip` or `*z` can all be used to access the first element of the array `z`

How To Access Next Element Using Pointer?

- What about accessing `z[1]` using pointers ?

Is it `*(ip+1)`?

- *Hmmm…*
- Since `ip` is an address, adding `1` will just point to the next byte
- But since the array consists of ints (which are more than 1 byte), `ip+1` will still point to a certain part of the first element?

Pointer Arithmetic

- Addition and subtraction can be performed on pointers
- Suppose :

```
data_type *name;
```

```
name + k
```

Evaluated as

```
name + k*sizeof(data_type)
```

```
name - k
```

Evaluated as

```
name - k*sizeof(data_type)
```

Pointers and Arrays (3)

- **Arrays in C are pointed to**, i.e. the variable that you declare for the array is actually a **fixed pointer** to the first array element

- Example:

```
int z[10] = {1, 2, 3};
```

  - `z` is a fixed pointer, it points to the address of the first element `z[0]`
  - In other words, `z == &z[0]`
  - In general, `z+i == &z[i]`

Pointers and Arrays (4)

- Array elements are usually accessed using [ ] (with the index)

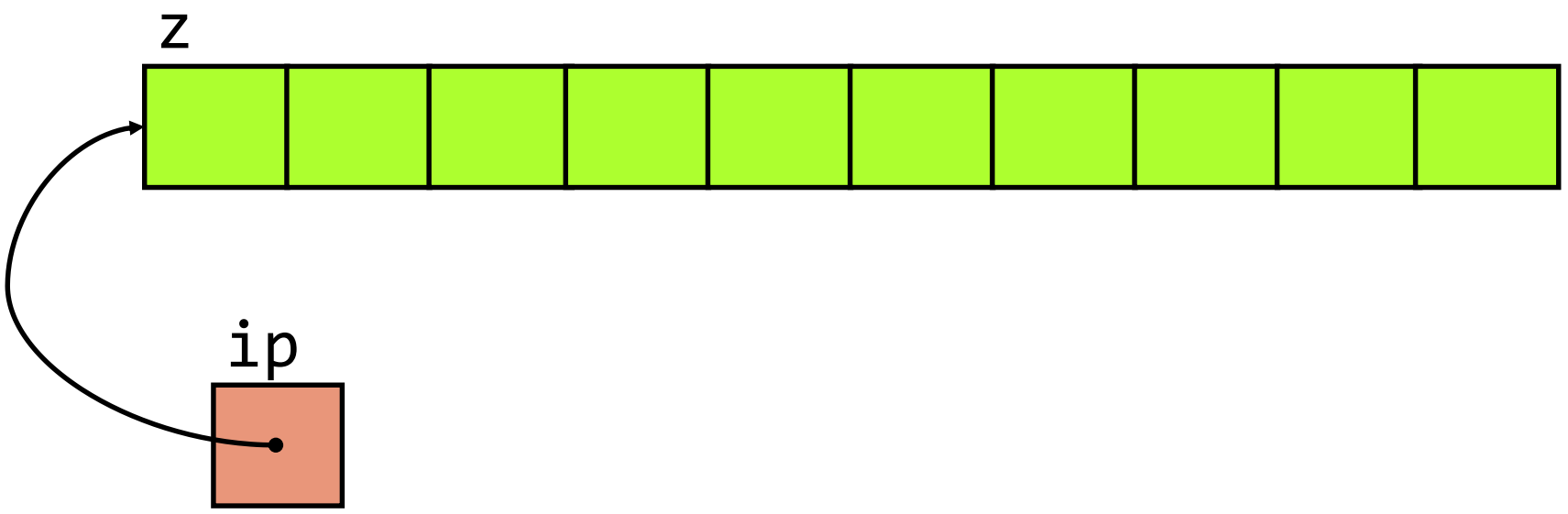
- Pointers can also be used to access array elements

```
int z[10], *ip;
ip = &z[0];
```

  - `z[i]`, `ip[i]`, `*(z+i)`, or `*(ip+i)` can all be used to access the `i`*th* element of the array `z`

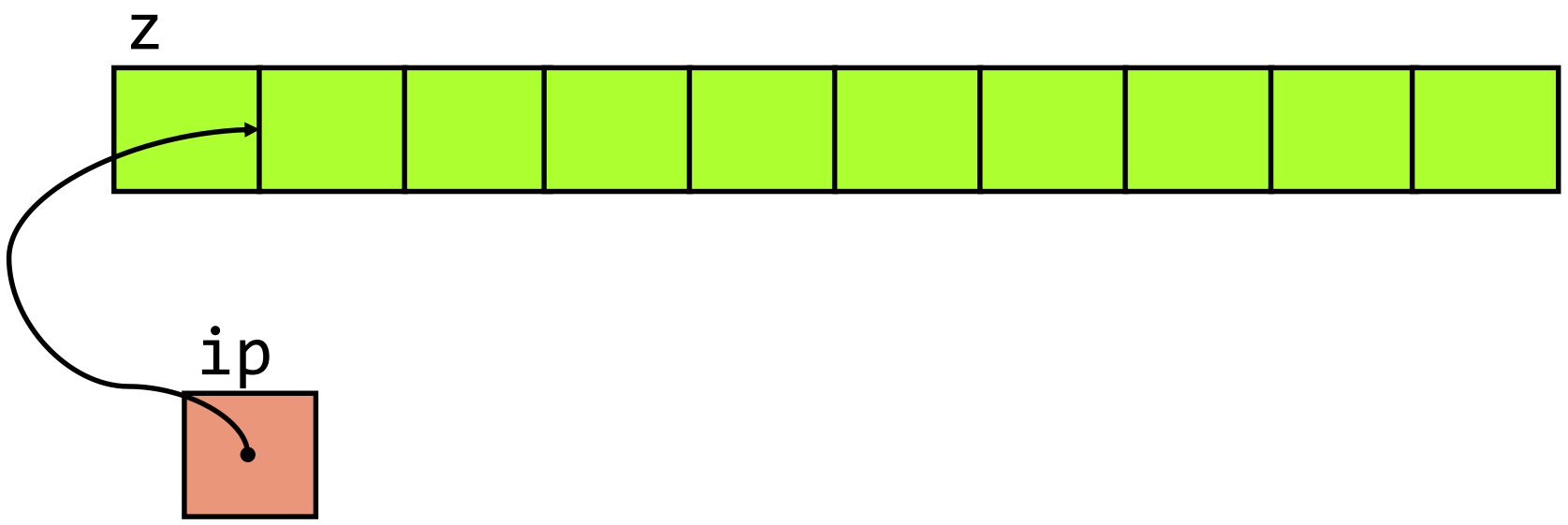
Graphical Illustration

```
int z[10], *ip;
ip = &z[0];
ip++; // ip = ip + 1
```



Graphical Illustration

```
int z[10], *ip;
ip = &z[0];
ip++; // ip = ip + 1
```



Traversing Arrays Using Pointers

The usual way to iterate over arrays:

```
int a[] = { ... };
int len = sizeof(a)/sizeof(int);
for(int i = 0; i < len; i++) {
      /* Do something about a[i] */
}
```

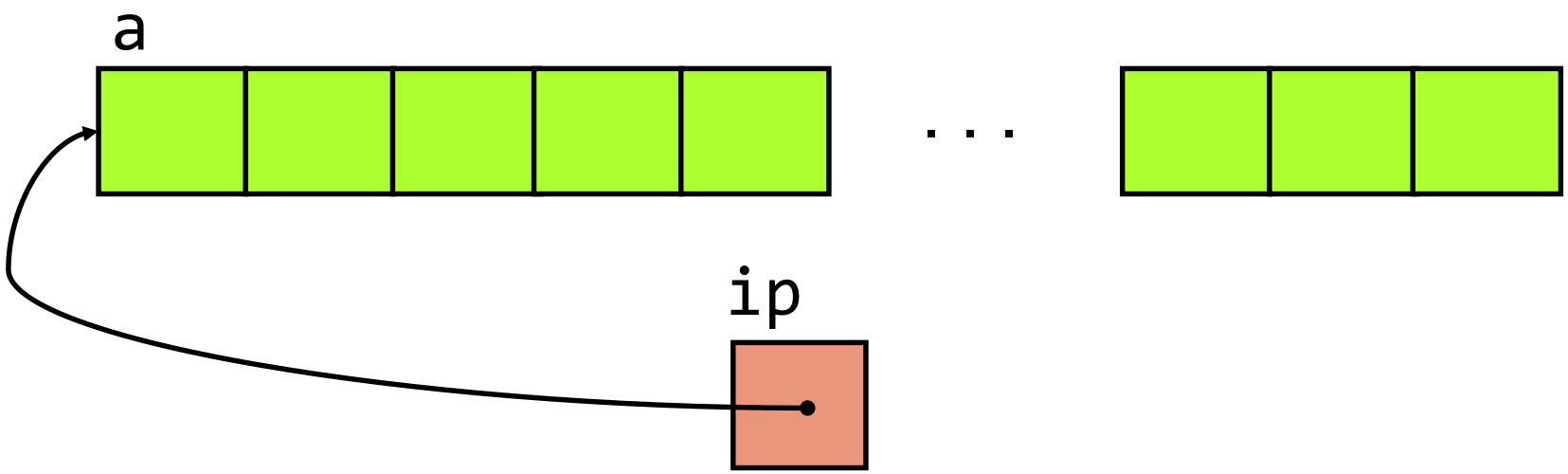
Using pointers:

```
int a[] = { ... };
int len = sizeof(a)/sizeof(int);
for(int *ip = a; ip < a + len; ip++) {
      /* Do something about *ip */
}
```

Traversing Arrays Using Pointers

```
int a[] = { ... };
int len = sizeof(a)/sizeof(int);
for(int *ip = a; ip < a + len; ip++) {
      /* Do something about *ip */
}
```

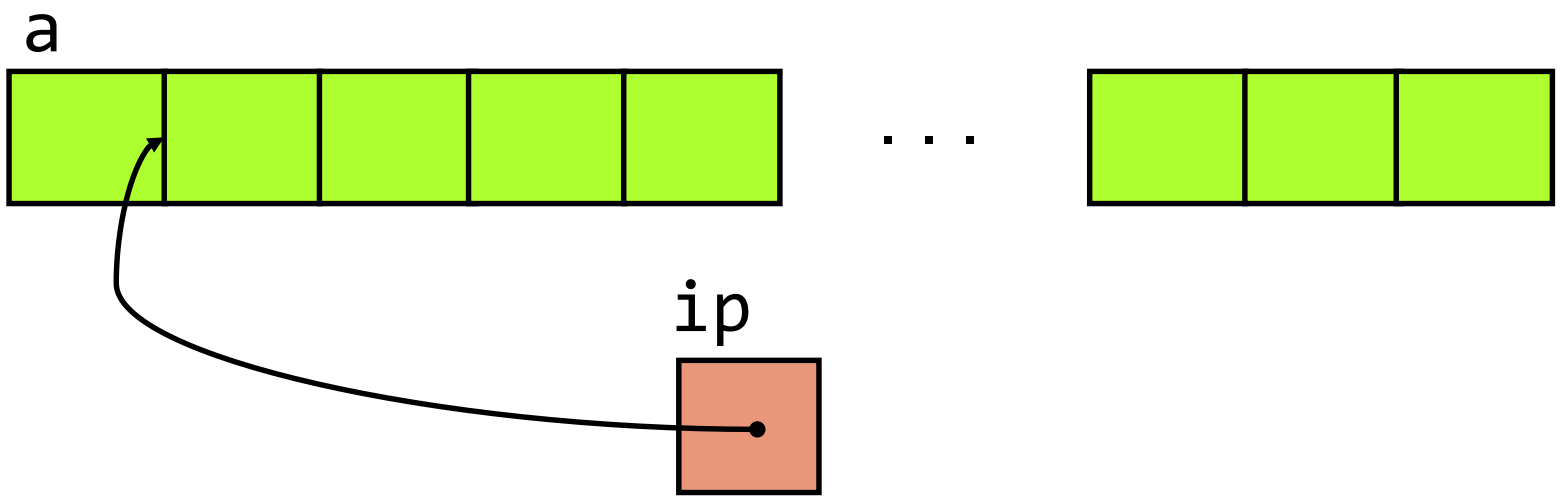
1st iteration:



Traversing Arrays Using Pointers

```
int a[] = { ... };
int len = sizeof(a)/sizeof(int);
for(int *ip = a; ip < a + len; ip++) {
      /* Do something about *ip */
}
```

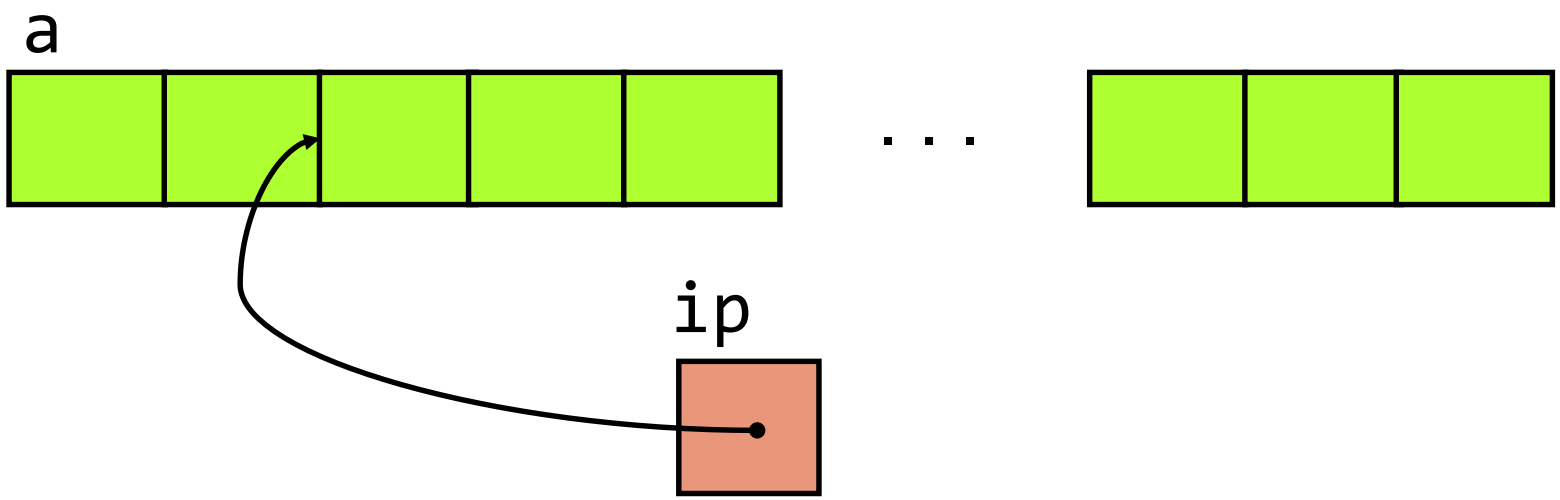
2nd iteration:



Traversing Arrays Using Pointers

```
int a[] = { ... };
int len = sizeof(a)/sizeof(int);
for(int *ip = a; ip < a + len; ip++) {
      /* Do something about *ip */
}
```

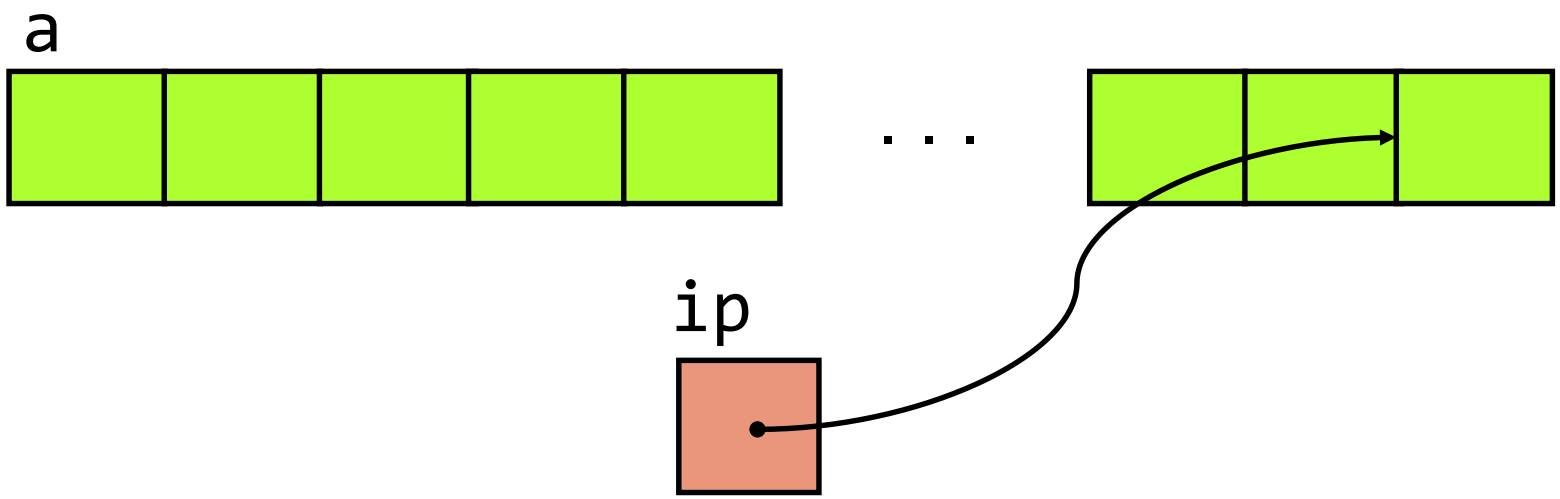
3rd iteration:



Traversing Arrays Using Pointers

```
int a[] = { ... };
int len = sizeof(a)/sizeof(int);
for(int *ip = a; ip < a + len; ip++) {
      /* Do something about *ip */
}
```

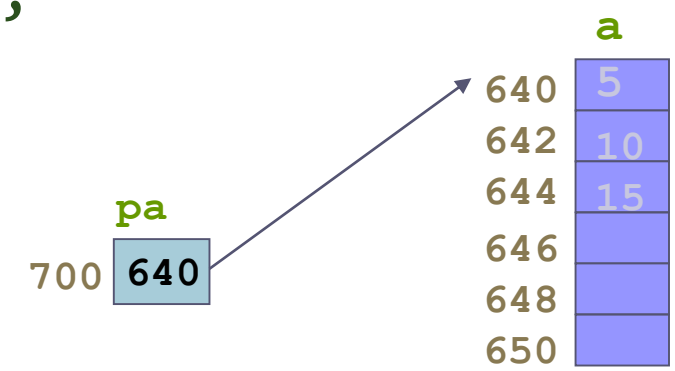
(len-1)th iteration:



Pointer Arithmetic

Assume short is 2 bytes, and pointer variable (address size) is 4 bytes.

```
short a[10]={5, 10, 15, …};
short *pa;
int i=5;
pa = &a;
```



Variable	Address	Value
a	640	5
	642	10
	644	15
	…	…
pa	700	640

Questions:

Expression	Value	Note
pa+1		
pa+3		
pa+i		
*pa+1		
*(pa+1)		
pa[2]		

A Note on Operator Precedence

Slight correction:

These only refer to *prefix* ++ and --

Postfix ++ and -- has level 1 precedence, i.e., the same as (), [], -> and .

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * (*type*) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left
,	left to right

Increment and Indirection Together

- Suppose

```
int *ip;
int i;
```

- What does `i = *ip++` mean?
  - Since postfix ++ has higher precedence than *, the RHS expression evaluates to `*(ip++)` which means

```
i = *ip; ip = ip + 1;
```

Increment and Indirection Together

- Suppose

```
int *ip;
int i;
```

- What does `i = *++ip` mean?
  - Both prefix ++ and * have same precedence, so associativity (right to left) is applied on RHS yielding `*(++ip)` which means

```
ip = ip + 1; i = *ip;
```

Increment and Indirection Together

- Suppose

```
int *ip;
int i;
```

- How to increment the value of whatever `ip` points to?

```
(*ip)++;
```

Pointer Types

- Pointer variables are generally of the same size, but it is **inappropriate** to assign an address of one type of pointer variable to a different type of pointer variable
- 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)

Casting Pointers

- When assigning a memory address of a variable of one type to a pointer that points to another type, it is best to use the cast operator to indicate the cast is intentional (this will remove the warning).
- Example:

```
int V = 101;
float *P = (float *) &V;
/* Casts int address to float * */
```

  - Removes warning, but is still unsafe to do this !!! You must know what you are doing when casting pointers!

General (void) Pointer

- A `void *` is considered to be a **general pointer**, it can point to any type of pointer variable
- No cast is needed to assign an address to a `void *` or from a `void *` to another pointer type
- Example:

```
int V = 101;
void *G = &V; /* No warning */
float *P = G; /* No warning, still unsafe */
```

- Certain library functions return `void *` results

Pointer to Pointer

- A pointer can also be made to point to a pointer variable (but the pointer must be of a type that allows it to point to a pointer)

- Example:

```
int V = 101;
int *P = &V;  /* P points to int V */
int **Q = &P; /* Q points to int pointer P */

printf("%d %d %d\n", V, *P, **Q);
  /* prints 101 3 times */
```

Strings and Pointers

- Recall:
  - A string in C is an array of chars terminated by the null character
  - We can use a pointer to point to an array
- **A char pointer can be used to point to a string**

```
char str[] = "Hello, world";
char *vstr = str1;
char *lstr = "Hello, world";
```

`vstr` points to a string variable

`lstr` points to a string literal

```
vstr[0] = 'h';
lstr[0] = 'h';
```

Allowed since `vstr` points to a string variable

Not allowed since `lstr` points to a string literal

Strings ❤ Pointers

```
int strlen (char *s)
{
  int n;
  for(n=0; *s!='\0'; s++)
    n++;
  return n;
}
```

```
int strcmp(char *s, char *t)
{
  int i;
  for(i=0;s[i]==t[i];i++)
    if(s[i] == '\0')
      return 0;
  return s[i] - t[i];
}
```

```
void strcpy(char *s, char *t)
{
  int i = 0;
  while((s[i]=t[i]) != '\0')
    i++;
}
```

Notice in the second `strcmp()` and second and third `strcpy()`, the use of pointers to iterate through the strings

```
int strcmp(char *s, char *t)
{
  for(;*s == *t; s++,t++)
    if (*s == '\0')
      return 0;
  return *s - *t;
}
```

```
void strcpy(char *s, char *t)
{
  while((*s=*t) != '\0') {
    s++; t++;
  }
}
```

The conciseness of the last `strcmp()` and `strcpy()` make them hard to understand

```
void strcpy(char *s, char *t)
{
  while((*s++=*t++) != '\0');
}
```

Next Lecture

- More Pointers
- Storage Classes
- C Process Layout