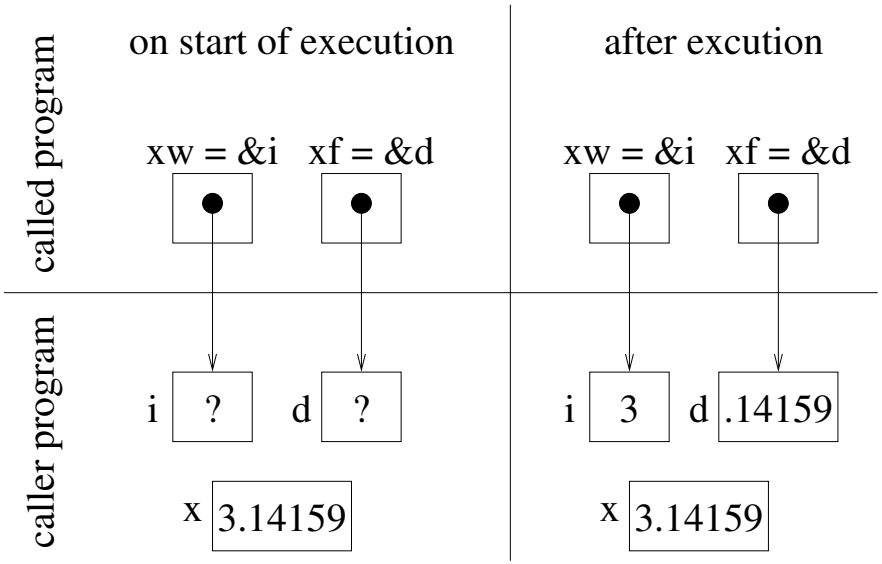
Pointers as Arguments

How it Works



Pointers as Arguments

Nonpointer Arguments for Pointers Parameters

- Failing to pass pointer, when one is expected, can be disastrous.
- Function treats arguments as pointers, and uses values of arguments as indirection.
- It, therefore, causes change in contents of an unknown memory location instead of the contents relevant to the arguments of the caller program.

Pointers as Arguments

Protecting Pointer Arguments

- Pointer arguments can be protected against accidental changes by putting qualifier `const`.

```
void f(const int *p) {
    int j;
    *p = 0; // WRONG
    p = &j; // PERMITTED
}
```

```
void g(int * const p) {
    int j;
    *p = 0; // PERMITTED
    p = &j; // WRONG
}
```

```
void h(const int * const p) {
    int j;
    *p = 0; // WRONG  (object is protected)
    p = &j; // WRONG  (pointer is protected)
}
```

Pointers as Arguments

Pointer as Return Types

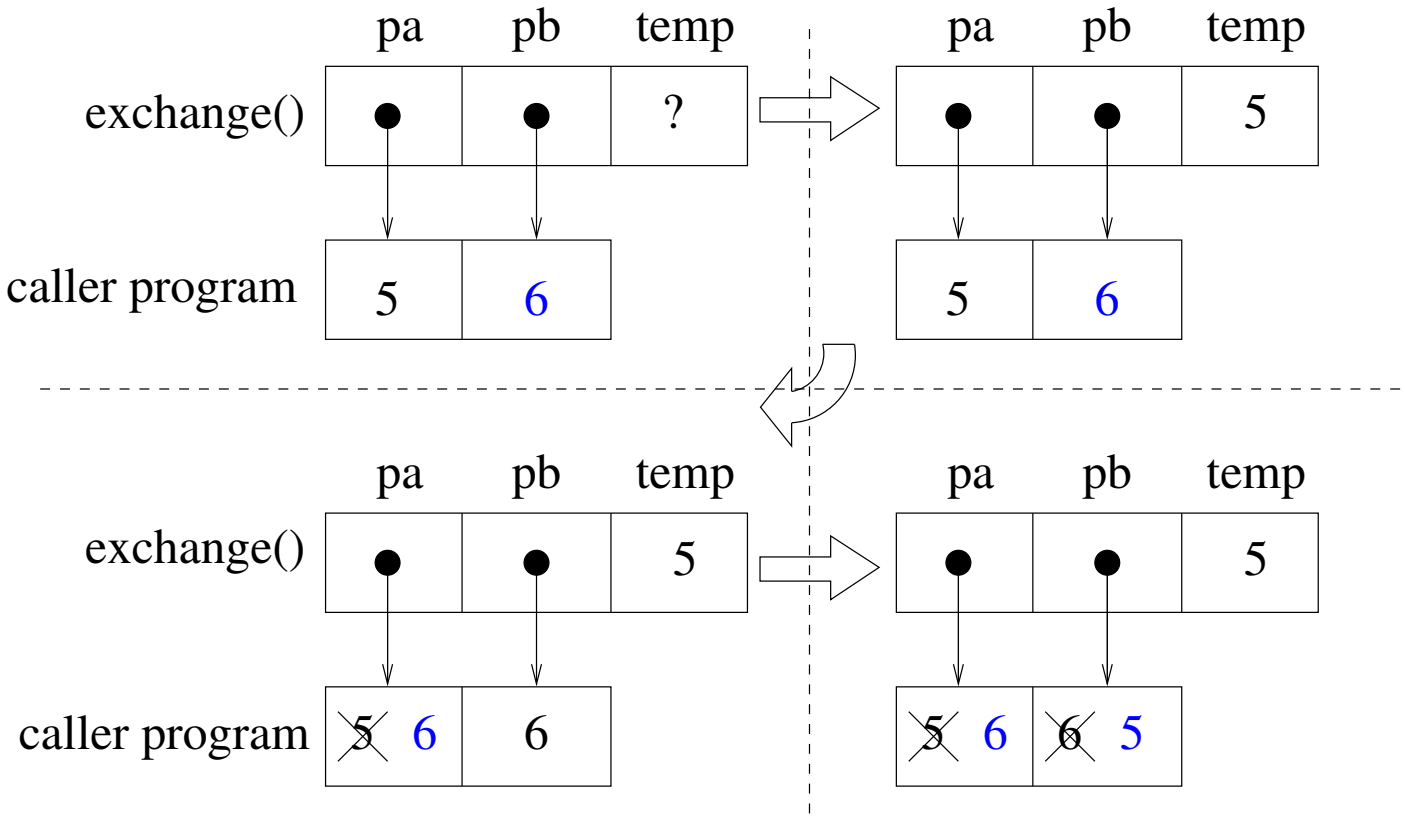
- A function may return pointer type.

```
int *max(int *a , int *b) {
     if (*a > *b)
          return a;
     else
          return b;
}

int main() {
     int *p, i, j;
     ...
     p = max(&i, &j);
}
```

Pointers

Exchange Two Values



Pointers

Exchange Two Values

```
#include <stdio.h>
void exchange(int *ip1, int *ip2) {
    int temp;

    temp = *ip1;
    *ip1 = *ip2;
    *ip2 = temp;
}
int main() {
    int i = 10, j = 20;

    printf("Initially i = %d  and j = %d\n", i, j);
    exchange(&i, &j);
    printf("Now i = %d  and j = %d\n", i, j);
}
```

Pointers

Void Pointers

- Extremely useful as a generic pointer type.
- It can be cast into any non-void pointer type.
- In other words, it is covers up for lack of function overloading in C.
- Eg. a pointer to `char`, `float`, `int`, etc. can match as argument against a parameter that is a void pointer in a function.

Pointers

Void Pointers

```
void generic(void *a, int b) {
    if(b == 1) {
        printf("Received integer pointer\n");
        printf("%d\n",*(int*)a);    // casting necessary
    }
    else if(b == 2) {
        printf("Received character pointer\n");
        printf("%c\n",*(char*)a);
    }
    else if(b == 3) {
        printf("Received float pointer\n");
        printf("%.1f\n",*(float*)a);
    }
}
```

Pointers

Void Pointers

```
#include <stdio.h>
//  Code for generic(void *a, int b)
int main() {
    int i = 15;
    char c = 'Y';
    float f = 15.5;

    generic(&i, 1);
    generic(&c, 2);
    generic(&f, 3);
}
```

Pointers

Generic Exchange

```
void genericExchg(void *a, void *b, int c) {
    if(c == 1) {
        int temp = *(int *) a;
        *(int *) a = *(int *) b;
        *(int *) b = temp;
    }
    else if(c == 2) {
        char temp = *(char *) a;
        *(char *) a = *(char *) b;
        *(char *) b = temp;
    }
    else if(c == 3) {
        float temp = *(float *) a;
        *(float *) a = *(float *) b;
        *(float *) b = temp;
    }
}
```

Pointers

Generic Exchange

```
#include <stdio.h>
int main() {

    int i = 15, j = 25;
    char c = 'Y', d = 'N';
    float f = 15.5, g = 31.0;

    genericExchg(&i, &j, 1);
    printf("i = %d and j = %d\n", i, j);

    genericExchg(&c, &d, 2);
    printf("c = %c and d = %c\n", c, d);

    genericExchg(&f, &g, 3);
    printf("f = %.2f and g = %.2f\n", f, g);

}
```

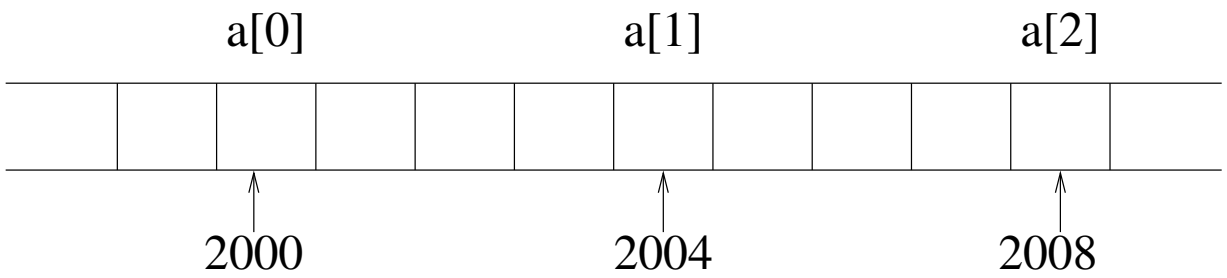
Pointer Arithmetic

Incrementing Pointer

- The contents of pointer variable (**lvalue** of a different variable) is treated like any other number.
- How arithmetic operations are defined on pointers?
- If a pointer to integer is defined then a block of 4 bytes reserved.
- So, for `int *ip`, the operation `ip + 1` adds 4 to `ip`'s contents.
- It points to next integer in memory location.
- `++ip` and `ip++` have same meaning.
- The pointer variable is incremented by `sizeof(type)`

Pointer Arithmetic

Arrays and Pointers



- An array defines a contiguous location of memory for its elements.
- By incrementing pointer to an array different elements of the array can accessed.
- `a + i` will be equal to address **2000 + (i * 4)**
- General formula: **addr(ptr + i) = addr(ptr) + sizeof(type) * i**

Pointer Arithmetic

Arrays and Pointers

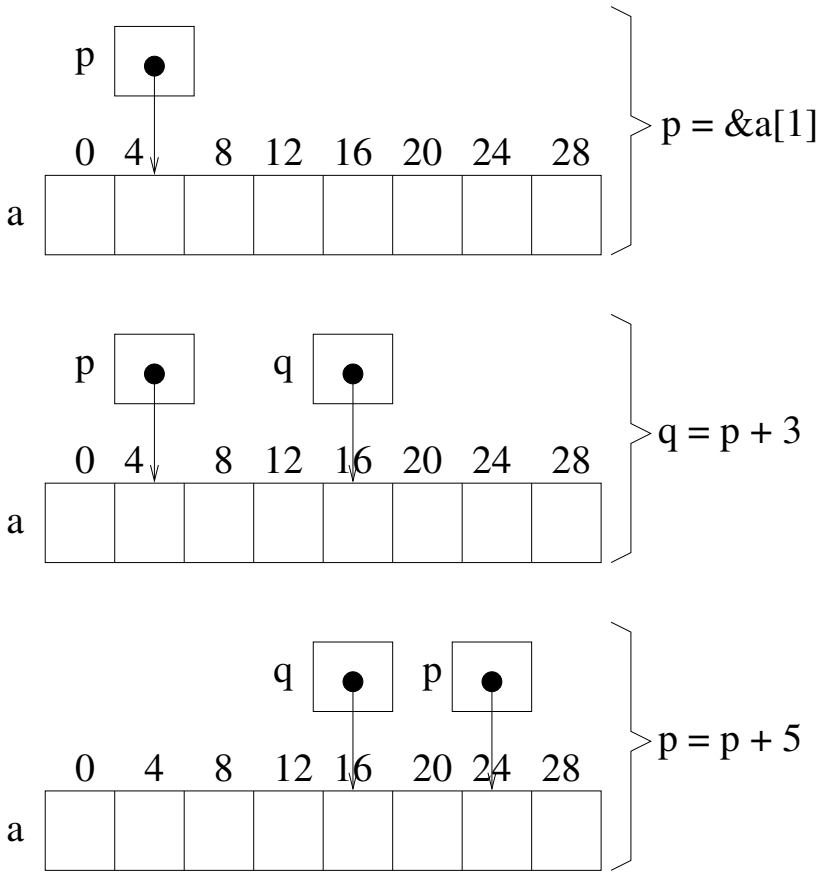
```
int a[10];
a = a + 1 // Not permited (a is a constant pointer)

int a[10];
int *ap;

ap = a + 4; // Permitted
```

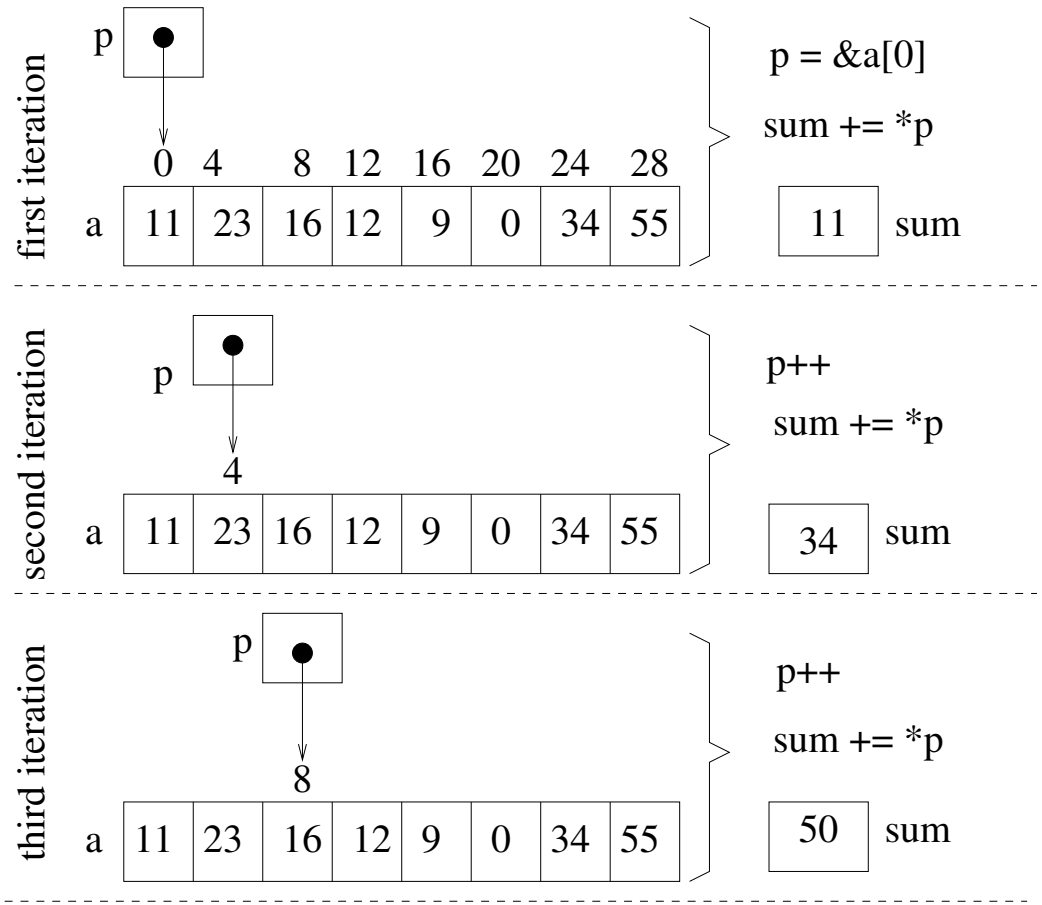
Pointer Arithmetic

Arrays and Pointers



Pointer Arithmetic

Arrays and Pointers



Pointer Arithmetic

Arrays and Pointers

- If `p` is pointing to `a[i] = k` then `*p++` should be `k` (* has greater precedence).

Expression	Meaning
*p++ or *(p++) or (*p)++	Increment p later: value is *p before increment
*++p or *(++p)	Increment p first: value is *p after increment
++*p or ++(*p)	Increment *p first: value is *p after increment