Program Organization and Pointer

Index

01. Program Organization

- Local and External Variables
- Block
- Scope

02. Pointer

- Pointer Variables
- Address and Indirection Operators
- Pointer Assignment
- Return with Pointer

01

Program Organization

Program Organization

Local and External Variables

Local variable

- It declared in the body of a function is regarded as local to the function

```
int sum_digits(int n)
{
    int sum = 0;   // local variable

    while (n > 0) {
        sum += n % 10;
        n /= 10;
    }

    return sum;
}
```

Program Organization

Local and External Variables

Default properties of local variables

- Automatic storage duration: storage is "automatically" allocated when the enclosing function is called and de-allocated when the function returns
- Block scope: a local variable is visible from its point of declaration to the end of the enclosing function body

```
int main()
{
    int variable1;            /* Automatic Storage Duration */
    int variable2 = 0;        /* Automatic Storage Duration */
    int array1[4];            /* Automatic Storage Duration */
    {
        int i;                /* Automatic Storage Duration */
    } /* Now, "i" is dead */
    return EXIT_SUCCESS;
} /* Now, "variable1", "variable2", "array1" are dead. */
```

Program Organization

Local and External Variables

Static storage duration

- using ***static*** before the declaration of variable makes the variable have static storage duration
- A variable with static storage duration has a permanent storage location, it retains its value throughout the execution of the program
- A static local variable still has block scope, and it's not visible to other functions

```c
void count(void)
{
  static int c = 1;
  printf("c = %d\n", c);
  c++;
}
for(int i = 0; i < 10; i++)
{
  count();
}
```

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

Program Organization

Local and External Variables

External variable

- Passing arguments is one way to transmit information to a function
- Functions can also communicate through external variables
  - Variables that are declared outside the body of any function
- External variables are sometimes known as global variables
- Properties of external variables
  - Static storage duration
  - File scope
- Having file scope means that an external variable is visible from its point of declaration to the end of the enclosing file

```c
#include <stdio.h>

int hello = 5;

void count(void)
{
  static int c = 1;
  printf("c = %d\n", c);
  c++;
}

int main(void)
{
  for(int i = 0; i < 10; i++)
  {
    count();
  }
  printf("%d\n", hello);

  return 0;
}
```

Program Organization

Local and External Variables

External variables are convenient when functions must share a variable or when a few functions share a large number of variables

In most cases, it's better for functions to communicate through parameters rather than by sharing variables

- If we change an external variable during program maintenance (by altering its type, say), we'll need to check every function in the same file to see how the change affects it
- If an external variable is assigned an incorrect value, it may be difficult to identify the guilty function
- Functions that rely on external variables are hard to reuse in other programs

Program Organization

Block

Block

- By default, the storage duration of a variable declared in a block is automatic; storage for the variable is allocated when the block is entered and deallocated when the block is exited
- The variable has block scope; it can't be referenced outside the block
- A variable that belongs to a block can be declared static to give it static storage duration

```
if (i > j)
{
    // swap values of i and j
    int temp = i;
    i = j;
    j = temp;
}
```

Program Organization

Block

The body of a function is a block, and the blocks are also useful inside a function body when we need variables for temporary use

Advantages of declaring temporary variables in blocks:

- Avoids cluttering declarations at the beginning of the function body with variables that are used only briefly
- Reduces name conflicts

Program Organization

Scope

In a C program, the same identifier may have several different meanings

C's scope rules enable the programmer (and the compiler) to determine which meaning is relevant at a given point in the program

The most important scope rule: When a declaration inside a block names an identifier that's already visible, the new declaration temporarily "hides" the old one, and the identifier takes on a new meaning

At the end of the block, the identifier regains its old

```c
int main()
{
    int i = 1;

    if(1)
    {
        int i = 2;
        printf("i = %d\n", i);
    }
    printf("i = %d\n", i);

    return 0;
}
```

```
i = 2
i = 1
```

Program Organization

Block

The body of a function is a block, and the blocks are also useful inside a function body when we need variables for temporary use

Advantages of declaring temporary variables in blocks:

- Avoids cluttering declarations at the beginning of the function body with variables that are used only briefly
- Reduces name conflicts

Program Organization

Block

```
int i ;                  /* Declaration 1 */

void f(int i )           /* Declaration 2 */
{
  i = 1;
}

void g(void)
{
  int i = 2;             /* Declaration 3 */

  if (i > 0) {
    int i ;              /* Declaration 4 */

    i = 3;
  }

  i = 4;
}

void h(void)
{
  i = 5;
}
```

Program Organization

An Example

Write a program to guess a random number using external variables

```
Guess the secret number between 1 and 100.
A new number has been chosen.
Enter guess: 50
The number is 50-100
Enter guess: 75
The number is 50-75
Enter guess: 88
Re-enter guess (50-75): 60
The number is 50-60
Enter guess: 59
You won in 5 guesses!



Play again? (Y/N)
```

- time (from <time.h>) - returns the current time
  - time(NULL)
- srand (from <stdlib.h>) - initializes C's random number generator
  - srand((unsigned) time(NULL))
- rand (from <stdlib.h>) - produces an apparently random number
  - rand()

02

Pointer

Pointer

Pointer Variables

Main memory is divided into bytes and each byte is capable of storing eight bits of information

Each byte has a unique ***address***

If there are n bytes in memory, the addresses as numbers that range from 0 to n-1 could be thought as

Address	Content							
0	0	1	1	1	0	1	1	0
1	0	0	0	1	1	1	1	0
. . .	. . .							
n-1	1	1	0	1	1	1	1	1

Pointer

Pointer Variables

Each variable in a program occupies one or more bytes of memory

The address of the first byte is determined as the address of the variable

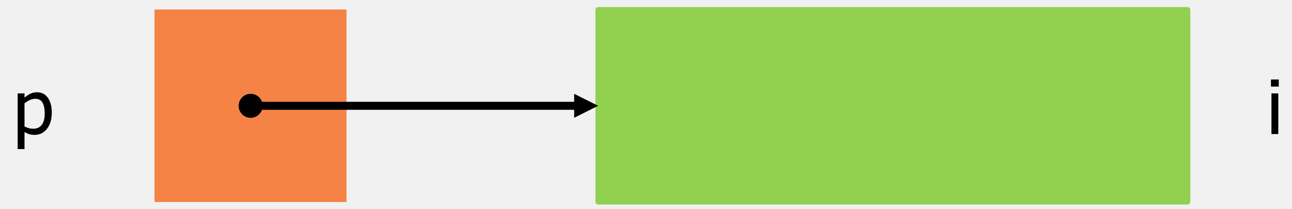
The address of i

Pointer

Pointer Variables

The address can be stored in special ***pointer variables***

If the address of a variable **i** is stored in the pointer variable **p**, we say that p "**points to**" i



When a pointer variable is declared, the asterisk must precede the variable name

```
int *p;
```

p is a pointer variable capable of pointing to objects of type int

The pointer variable might point to an area of memory

Pointer

Address and Indirection Operators

Every pointer variable points only to objects of a particular (***referenced***) type

```
int *p;        // points only to integers
double *q;     // points only to doubles
char *r;       // points only to characters
```

C provides a pair of operators designed for using with pointers

- To obtain the address of a variable, the **&** (**address**) operator is employed
- To access the value from a address, the ***** (**indirection**) operator is used

```c
int x = 5, y;
int *p;

p = &x;

printf("x's address = %d, value = %d\n\n\n\n", p, *p);
```

```
x's address = 6422296, value = 5
```

Pointer

Address and Indirection Operators

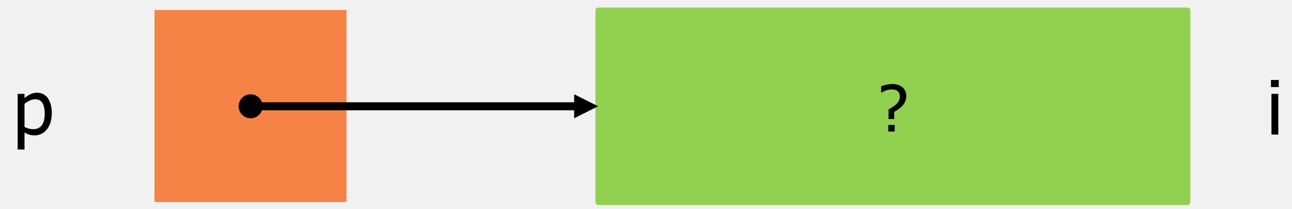
Address Operator

- It's crucial to initialize p before using it
- One way to initialize a pointer variable is to assign it the address of a variable such as

```
int x, *p;
p = &x;
```

```
int x;
int *p = &x;
```

```
int x, *p = &x;
```



Pointer

Address and Indirection Operators

Indirection Operator

- Once a pointer variable points to an object, the * operator can be used to access the value stored in the object

```
int x = 10, *p;
p = &x;
printf("%d\n", *p);

j = *&i;   // same as j = i;
```

As long as p points to i, *p is an alias for i, i.e.

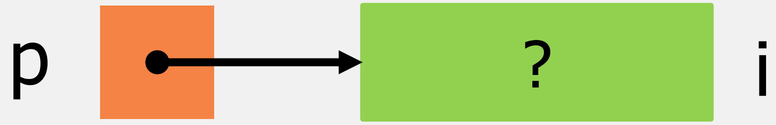
- *p has the same value as i
- Changing the value of *p is same as changing the value of i

Pointer

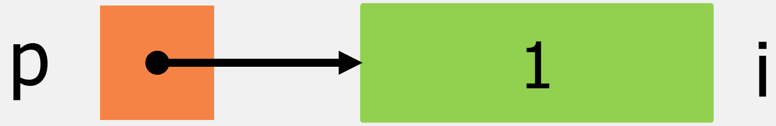
Address and Indirection Operators

Indirection Operator

```
p = &i;
```

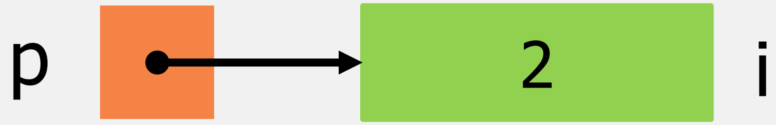



```
i = 1;
```




```
printf("%d\n", i);      //prints 1
printf("%d\n", *p);   // prints 1
```

```
*p = 2;
```




```
printf("%d\n", i);      //prints 2
printf("%d\n", *p);   // prints 2
```

i is variable and p points to i, which of the following expressions are aliases for i?

(a) *p (b) *&p (c) *i (d) *&i

(e) &p (f) &*p (g) &i (h) &*i

Pointer

Address and Indirection Operators

Applying the indirection operator to an uninitialized pointer variable causes undefined behavior

```
int *p;
printf("%d", *p);     // Wrong
```

Assigning a value to *p is particularly dangerous

```
int *p;
*p = 1;         // Wrong
```

Pointer

Address and Indirection Operators

Conversion specification: %p

```c
int x = 5, y;
int *p;

p = &x;

printf("x's address (D) = %d, (H) = %x, (P) = %p\n\n\n\n", p, p, p);
```

```
x's address (D) = 6422296, (H) = 61ff18, (P) = 0061FF18
```

6,422,296

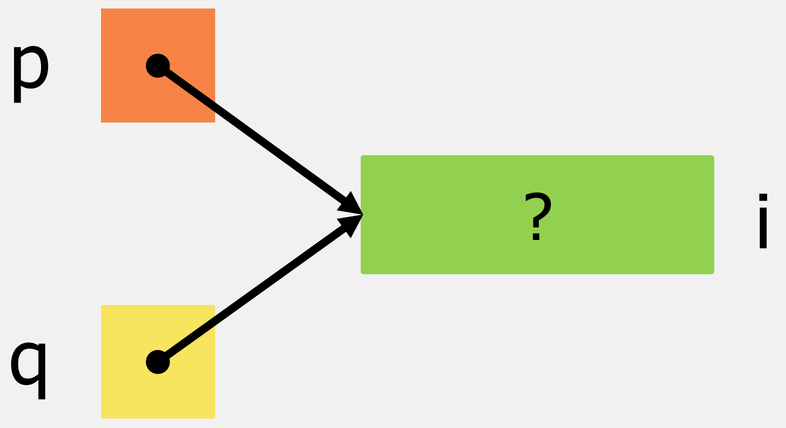
HEX	61 FF18
DEC	6,422,296
OCT	30 377 430
BIN	0110 0001 1111 1111 0001 1000

Pointer

Pointer Assignment

C allows the use of assignment operator to copy pointers of the same type

```
int i, j, *p, *q;
p = &i;
q = p;
```

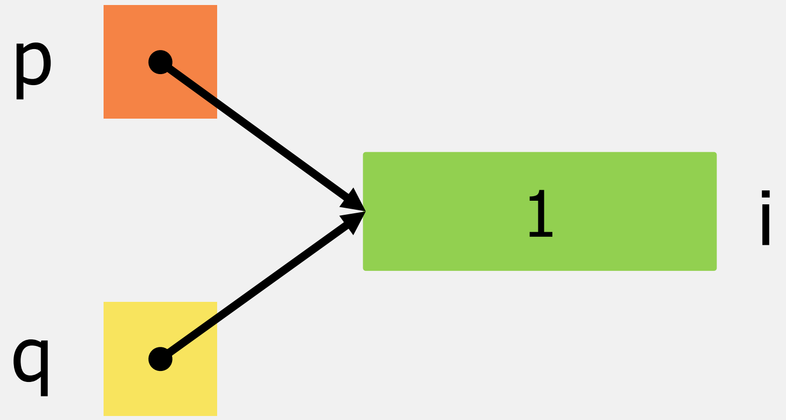


Pointer

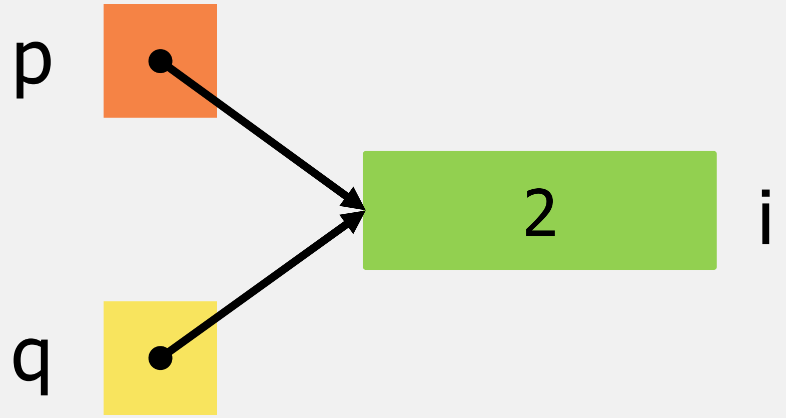
Pointer Assignment

If p and q both point to i, i can be changed by assigning a new value to either *p or *q

```
*p = 1;
```




```
*q = 2;
```

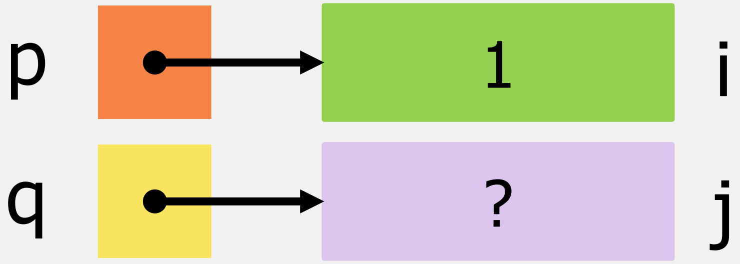


Pointer

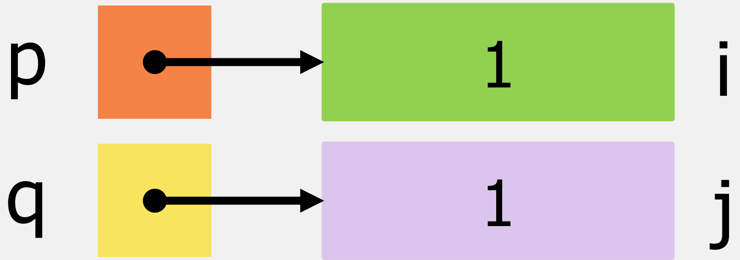
Pointer Assignment

Be careful not to confuse q = p with *q = *p

```
p = &i;
q = &j;
i = 1;
```




```
*q = *p
```



If i is an int variable and p and q are pointers to int, which of the following assignments are legal?

(a) p = i	(b) p = &q	(c) p = *q
(d) *p = &i	(e) p = *&q	(f) *p = q
(g) &p = q	(h) p = q	(i) *p = *q

Pointer

Pointer Assignment

New definition of decompose

```
void decompose(double x, long *int_part, double *frac_part)
{
    *int_part = (long) x;
    *frac_part = x - *int_part;
}
```

```
void decompose(double x, long *int_part, double *frac_part);
void decompose(double, long *, double *);
```

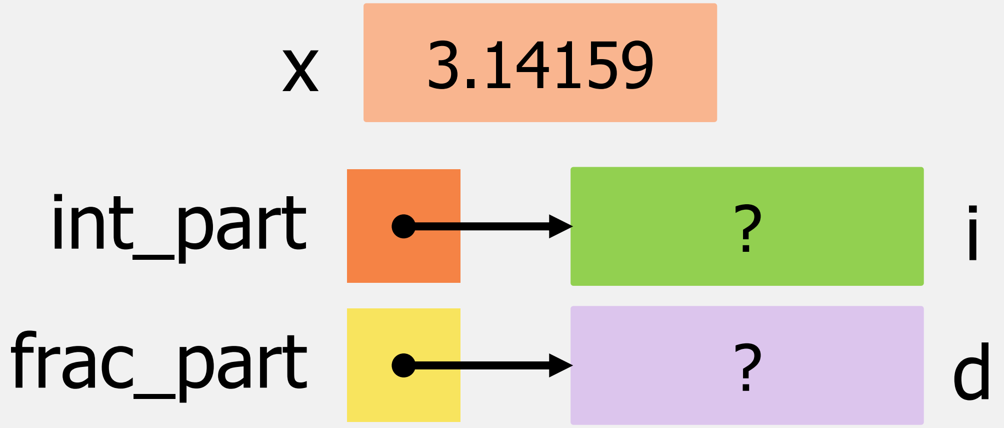
Pointer

Pointer Assignment

When calling the decompose

```
void decompose(double x, long *int_part, double *frac_part)
{
      *int_part = (long) x;
      *frac_part = x - *int_part;
}


int i, d;
decompose(3.14159, &i, &d);
```



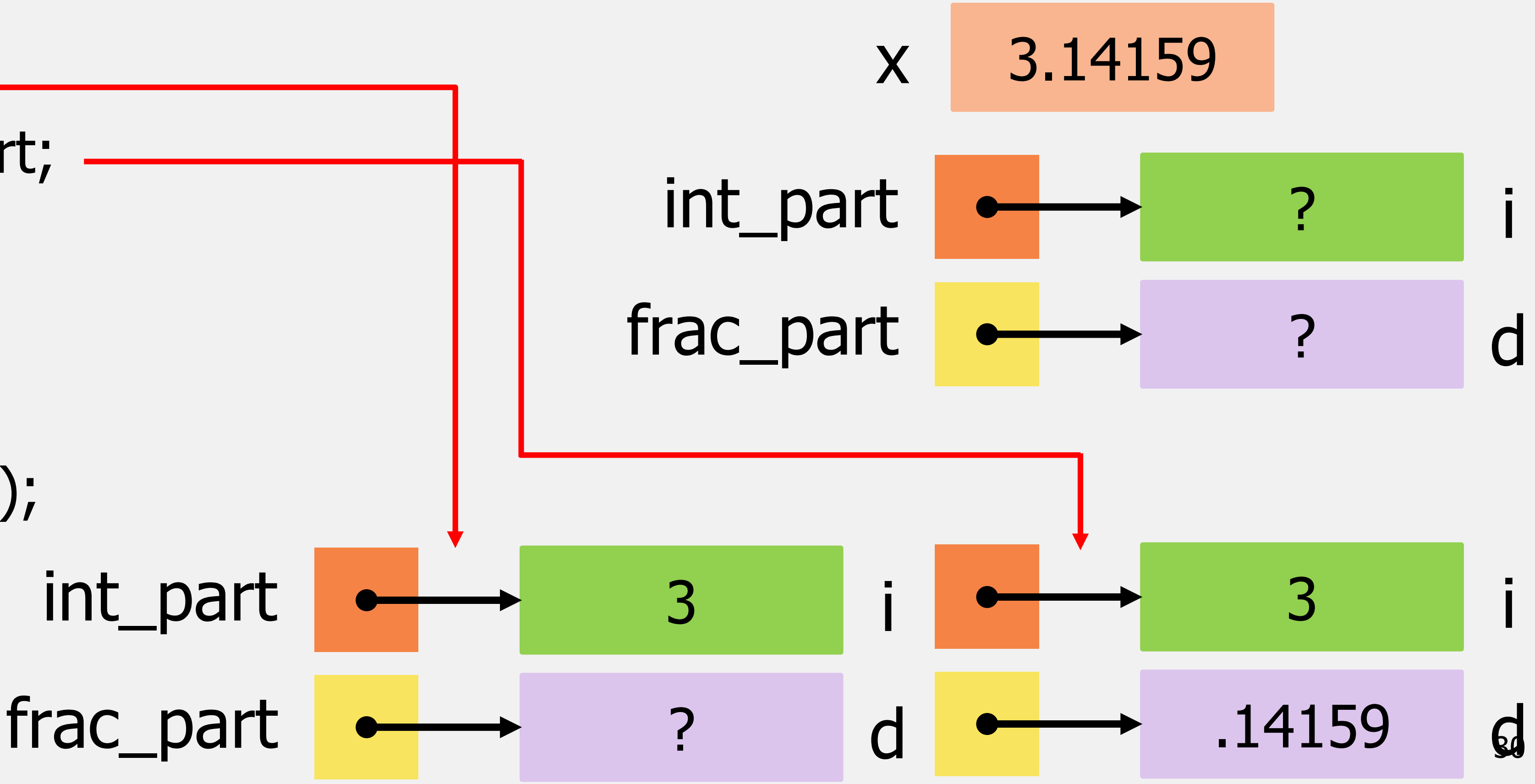
Pointer

Pointer Assignment

When calling the decompose

```
void decompose(double x, long *int_part, double *frac_part)
{
    *int_part = (long) x;
    *frac_part = x - *int_part;
}
```

```
int i, d;
decompose(3.14159, &i, &d);
```



Pointer

Pointer Assignment

Recall the *scanf* function

```
int i;
…
scanf("%d", &i);
```

Without the &, scanf would be supplied with the value of i

However, it's not always true that scanf needs the & operator

```
int i, *p;
   …
p = &i;
scanf("%d", p);        scanf("%d", &p);   // Wrong
```

Pointer

Pointer Assignment

Write a program to exchange the values of the variables using swap function

```
Enter 2 numbers for x and y: 25 6
After exchange
x = 6, y = 25
```

Pointer

Return with Pointer

Using const to protect argument

- when a argument is a pointer to a variable x, we normally assume that x will be modified

```
f(&x);
```

- It's possible, though, that f merely needs to examine the value of x, not change it
- The reason for the pointer might be efficiency: passing the value of a variable requires a large amount of storage

Pointer

Return with Pointer

Using const to protect argument

- Hence, const can be used to document that a function won't change an object whose address is passed to the function
- const goes in the parameter's declaration, just before the specification of its type:

```
void f(const int *p)
{
    *p = 0;   // Wrong
}
```

- Attempting to modify *p is an error that the compiler will detect

Pointer

Return with Pointer

```c
void f(const int *p)
{
    int j;
    p = &j;
    *p = 2;
}
```

```c
void g(int * const p)
{
    int j;
    p = &j;
    *p = 2;
}
```

```
test.c: In function 'f':
test.c:7:8: error: assignment of read-only location '*p'
     *p = 2;
        ^
```

```
test.c: In function 'g':
test.c:12:7: error: assignment of read-only parameter 'p'
     p = &j;
       ^
```

Pointer
Return with Pointer

The functions can return the pointers

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

int *p, i, j;
…
p = max(&i, &j);
```

Pointer

Return with Pointer

Important!! Never return a pointer to an automatic local variable such as

```
int *f(void)
{
    int i;
    ...
    return &i;
}
```

Why? Because the variable i won't exist after f returns

Pointer

Return with Pointer

Pointers can point to array element

If a is an array, then &a[i] is a pointer to the element i of a

```
int *find_middle(int a[], int n)
{
    return &a[n/2];
}
```

Pointer

Return with Pointer

Write a program to find the largest and smallest elements in an array

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
Smallest: 7
Largest: 102
```