C-8 Pointers

Why have pointers?

- Pointers allow different sections of code to share information easily. You can get the same effect by copying information back and forth, but pointers solve the problem better.
- Pointers enable complex "linked" data structures like linked lists and binary trees.
- The use of strings in C require a knowledge of pointers.

Addresses

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
        int a = 1, b = 2;
        printf(“a = %i; address of a = %u \n”, a, &a);
        printf(“b = %i; address of b = %u \n”, b, &b);
        return EXIT_SUCCESS;
}
```

output:

```
a = 1; address of a = 65524
b = 2; address of b = 65522
```

& is called an address operator

%u is conversion specifier for an unsigned integer

Pointer Declaration

Pointer – a variable that contains the memory address
of another variable.

A pointer must be defined to point to a specific **type** of variable.

An ***int*** pointer may not point to a ***double*** variable as an example.

Examples

Examples:

```
int a, b, *ptr;

float c, *fptr;
```

Reminders:

- * (asterisk) is called the *dereferencing* operator
  or *indirection* operator
- In a type declaration statement, the asterisk shows that the variable is being declared a pointer variable.

Example:

```
int a, b, *ptr;
```

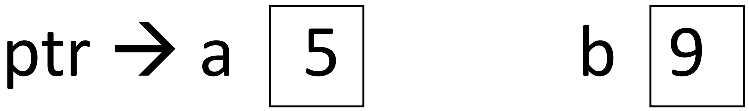
a $\boxed{?}$ b $\boxed{?}$ ptr $\boxed{?}$ →

```
int a, b, *ptr;
ptr = &a;
```

ptr → a $\boxed{?}$ b $\boxed{?}$

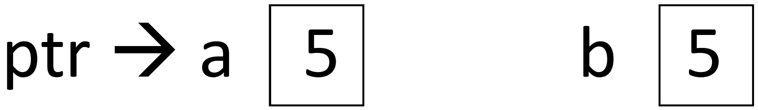
Now ptr points to variable a

int a = 5, b = 9, *ptr = &a;



----

b = *ptr;



Take the value from the variable-pointer points to, variable **a** which contains **5** and place it in the variable **b**

b = *ptr;
b = a;   both do same thing

int a = 5, b = 9, *ptr = &a;

ptr → a $\boxed{5}$ b $\boxed{9}$

***ptr = b;**

ptr → a $\boxed{9}$ b $\boxed{9}$

$$\left.\begin{array}{l} \text{*ptr = b;} \\ \\ \text{a = b;} \end{array}\right\} \text{ accomplish the same thing}$$

ptr - points to an address.

Ex: **int a, *ptr;**
ptr = &a;

*ptr - dereferences the pointer;
refers to the ***value*** in the address that ptr is pointing to

Ex: **a = 5;**
ptr = &a;

The value in *ptr is 5

```
int main(void) {
    int a = 1, b = 2, *A_ptr = &a;
    printf(“a = %i; address of a = %u \n”, a, &a);
    printf(“b = %i; address of b = %u \n”, b, &b);
    printf(“A_ptr = %u; address of A_ptr = %u \n”, A_ptr, &A_ptr);
    printf(“A_ptr points to the value %i \n”, *A_ptr);
    return EXIT_SUCCESS;
}
```

***output*:**

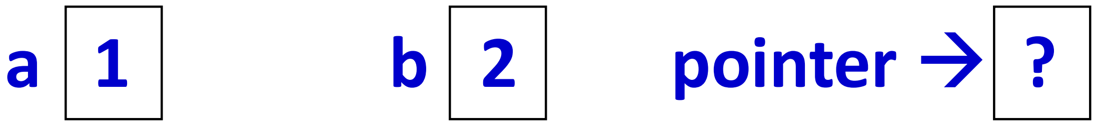
```
        a = 1; address of a = 65524
        b = 2; address of b = 65522
        A_ptr = 65524; address of A_ptr = 65520
        A_ptr points to the value 1
```

Give a memory snapshot after this set of statements is executed

```
int a = 1, b = 2, *pointer;
pointer = &b;
```

After the first line, the picture is:

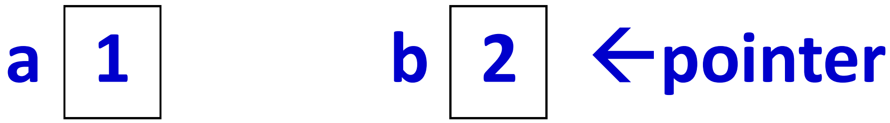
int a = 1, b = 2, *pointer;



a 1 b 2 pointer → ?

After the second line of code, the picture is:

```
int a = 1, b = 2, *pointer;
pointer = &b;
```



a 1 b 2 ←pointer

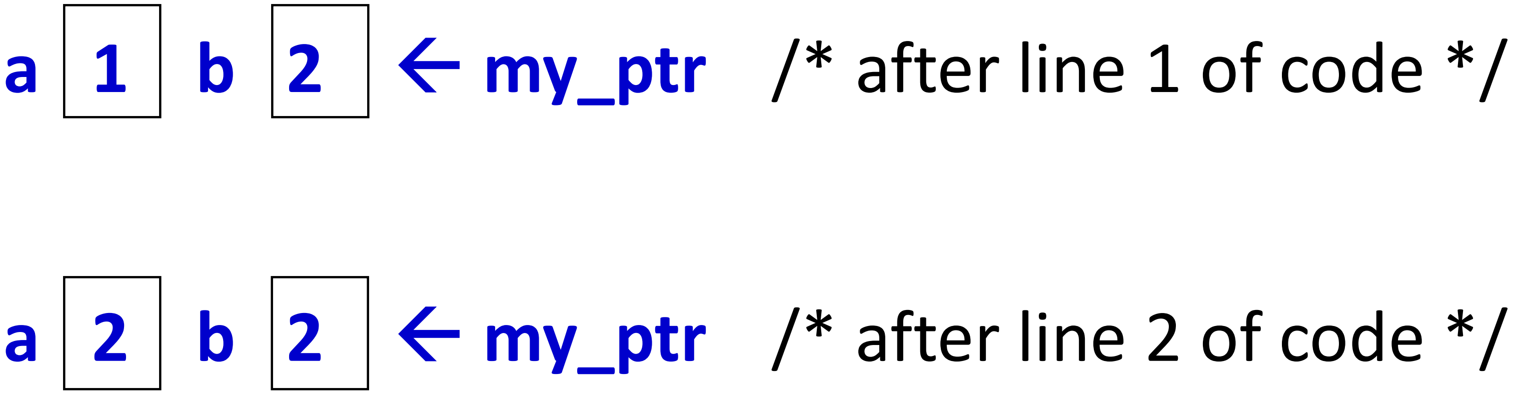
pointer contains the address of b

***pointer** contains the value of 2

Give a memory snapshot after this set of statements is executed

int a = 1, b = 2, *my_ptr = &b;

a = *my_ptr;



a 1 **b** 2 ← **my_ptr** /* after line 1 of code */

a 2 **b** 2 ← **my_ptr** /* after line 2 of code */

Give a memory snapshot after this set of statements is executed

```
int a = 1, b = 2, c = 5, *ptr = &c;
b = *ptr;
*ptr = a;
```

a $\boxed{1}$ b $\boxed{2}$ c $\boxed{5}$ ← ptr /* after line 1 of code */

a $\boxed{1}$ b $\boxed{5}$ c $\boxed{5}$ ← ptr /* after line 2 of code */

a $\boxed{1}$ b $\boxed{5}$ c $\boxed{1}$ ← ptr /* after line 3 of code */

Give a memory snapshot after this set of statements is executed

```
int a = 1, b = 2, c = 5, *ptr;
ptr = &c;
c = b;
a = *ptr;
```

a $\boxed{1}$ b $\boxed{2}$ c $\boxed{5}$ ptr→ $\boxed{?}$ /* after 1st line */

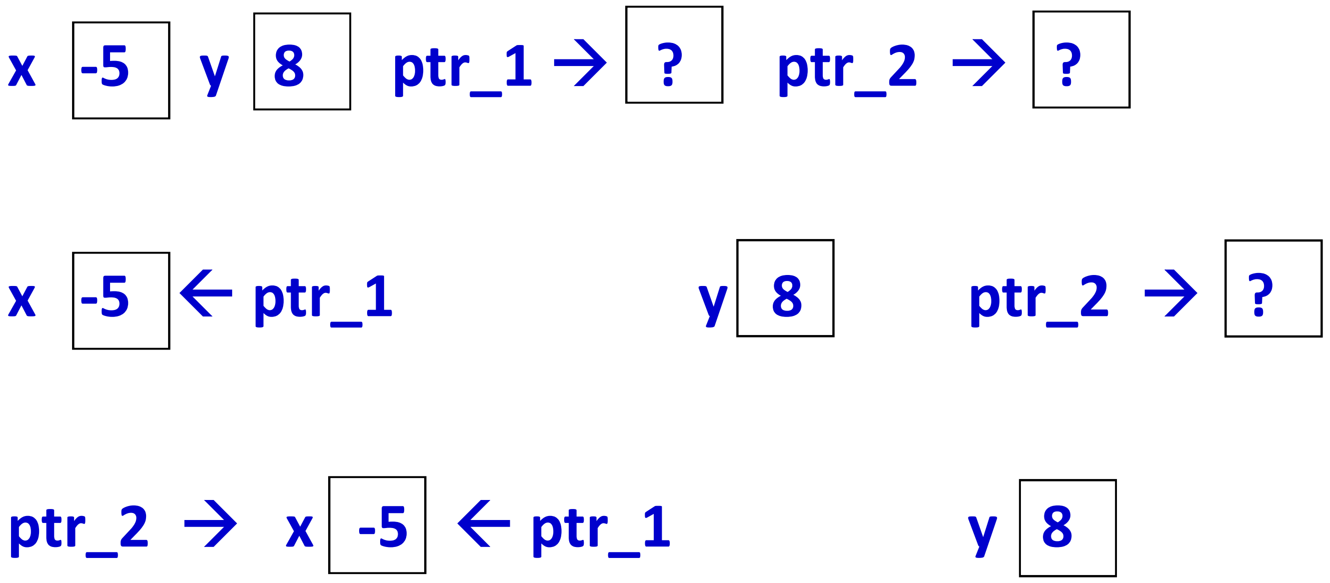
a $\boxed{1}$ b $\boxed{2}$ c $\boxed{5}$ ← ptr /* after 2nd line */

a $\boxed{1}$ b $\boxed{2}$ c $\boxed{2}$ ← ptr /* after 3rd line */

a $\boxed{2}$ b $\boxed{2}$ c $\boxed{2}$ ← ptr /* after 4th line */

A pointer can point to only one location,
but several pointers can point to the same location.

```
int x = -5, y = 8, *ptr_1, *ptr_2;
ptr_1 = &x;
ptr_2 = ptr_1;
```



x -5 y 8 ptr_1 → ? ptr_2 → ?

x -5 ← ptr_1 y 8 ptr_2 → ?

ptr_2 → x -5 ← ptr_1 y 8

FILE Pointers

File Pointer – a special pointer that holds the starting address of file.

```
FILE * sensor1;
sensor1 = fopen(“sensor1.dat”, “r”);
```

sensor1 is a pointer variable

```
fscanf(sensor1, “%f %f”, &t, &motion);
```

Read data from the file pointed to by **sensor1**

Pointer Address Arithmetic

Pointer Address Arithmetic

- Arithmetic operations can be performed on pointers
  - Increment/decrement pointer (**++** or **−−**)
  - Add an integer to a pointer(**+** or **+=** , **−** or **−=**)
  - Pointers may be subtracted from each other
  - Operations meaningless unless performed on an array

Address Arithmetic #1:

A pointer can be assigned to another pointer of the same type.

```
int x, *p1, *p2;

p1 = &x;

p2 = p1;
```

Address Arithmetic #2:

An integer value can be added to or subtracted from a pointer.

ptr++;   increments the pointer to point to the next
value in memory;
only works correctly with arrays

Address Arithmetic #3:

A pointer can be assigned or compared to the integer zero, or equivalently,
to symbolic constant NULL which is in <stdio.h>.

```
if (ptr == NULL)
{
    printf(“Error \n”);
}
```

Address Arithmetic #4:

Pointers to elements of the same array can be subtracted or compared.

```
ptr -= 3;
…
if (ptr < ptr + 1)
```

Common Errors

int y, *ptr1, *ptr2;

The following are all **invalid** statements:

&y = ptr1; attempts to change the address of y

ptr2 = y; attempts to change ptr2 to a non-address value

*ptr1 = ptr2; attempts to move an address to an integer variable

ptr1 = *ptr2; attempts to change ptr1 to a non-address value

It is not allowed to mix pointers of different types.

This shows an int with an int pointer,
and a float with a float pointer, using correct procedure.

```
int a,  *ptr_a;

float b, *ptr_b;
```

Memory assignments for elements of **arrays** are guaranteed to be sequential.

We can use a pointer to reference each element of an array.

Assign a pointer to the first element of the array and then reference the elements of the array by incrementing or decrementing the pointer.

Examples:

```
int x[10], *ptr_x;

ptr_x = &x[0];

ptr_x++;        increment ptr_x to point to the next
                value in memory
```

More examples:

int x[10], *ptr_x = &x[0];

ptr_x += 1; increment ptr_x to point to the next value in memory

ptr_x = &x[1]; ptr_x is assigned the address of x[1]

ptr_x += k; ptr_x is assigned the address k values past the one it was pointing to

Give memory snapshots after this set of statements is executed.

double x = 15.6, y = 10.2, *ptr1 = &y, *ptr2 = &x;

x $\boxed{15.6}$ ← ptr2   y $\boxed{10.2}$ ← ptr1

*ptr1 = *ptr2 + x;

so 15.6 + 15.6 = 31.2 hence

x $\boxed{15.6}$ ← ptr2   y $\boxed{31.2}$ ← ptr1

Give memory snapshots after this set of statements is executed.

int w = 10, x = 2, *ptr2 = &x;

w $\boxed{10}$ x $\boxed{2}$ ← ptr2

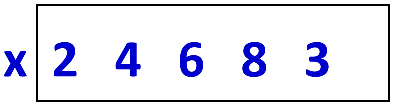
*ptr2 -= w;

so 2 – 10 = -8

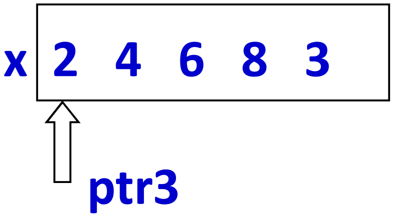
w $\boxed{10}$ x $\boxed{-8}$ ← ptr2

Give memory snapshots after this set of statements is executed.

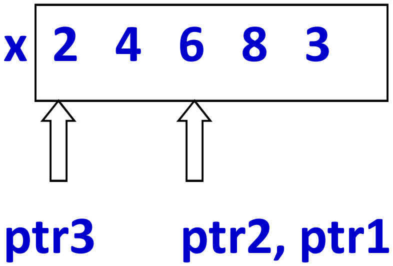
```
int x[5] = {2, 4, 6, 8, 3};
int *ptr1 = NULL, *ptr2 = NULL, *ptr3 = NULL;
ptr3 = &x[0];
ptr1 = ptr2 = ptr3 + 2;
```



ptr1 → NULL **ptr2 → NULL** **ptr3 → NULL**



ptr1 → NULL **ptr2 → NULL**



Give memory snapshots after this set of statements is executed.

```
int w[4], *first = NULL, *last = NULL;
first = &w[0];
last = first + 3;
```

w ? ? ? ? **first → NULL last → NULL**

---

w ? ? ? ? **last → NULL**

↑

first

---

w ? ? ? ?

↑ ↑

first last

Pointers and Arrays

Pointers and Arrays

int A[6] = {3, 2, 1, 4, 5, 6}, *ptr;

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
3	2	1	4	5	6

ptr = &A[0];

ptr + 2 refers to A[2]

ptr + 4 refers to A[4]

Pointers and Arrays

int A[6] = {3, 2, 1, 4, 5, 6}, *ptr=&A[0];

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
3	2	1	4	5	6

To sum the array:

```
sum = 0;
for (k = 0; k < 6; k++)
{
        sum += A[k];
}
```

Or ---------------------------------------

```
sum = 0;
for (k = 0; k < 6; k++)
{
        sum += *(ptr + k);
}
```

```
            ⬇        ⬇
int g[ ] = {2, 4, 5, 8, 10, 32, 78};
            0  1  2  3  4   5   6   → positions in array
```

```
int *ptr1 = &g[0];
int *ptr2 = &g[3];
```

What is the value of:

***g**

```
            ⬇        ⬇
int g[ ] = {2, 4, 5, 8, 10, 32, 78};
            0  1  2  3   4   5   6  → positions in array
```

```
int *ptr1 = &g[0];
int *ptr2 = &g[3];
```

What is the value of:

***g**

2 = answer

The name of an array acts like a pointer to the beginning of the array when the array name is missing the brackets [ ].

```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};
```

0 1 2 3 4 5 6 → positions in array

```
int *ptr1 = &g[0];
int *ptr2 = &g[3];
```

What is the value of: ***g + 1**

```
                ⬇        ⬇
int g[ ] = {2, 4, 5, 8, 10, 32, 78};
            0  1  2  3   4   5   6 → positions in array
```

```
int *ptr1 = &g[0];
int *ptr2 = &g[3];
```

What is the value of: ***g + 1**

3 = answer

Go to g, position zero.
Dereference getting the 2
Add 1 to the 2 and get 3

```
                ⬇       ⬇
int g[ ] = {2, 4, 5, 8, 10, 32, 78};
            0  1  2  3   4   5   6  → positions in array

int *ptr1 = &g[0];
int *ptr2 = &g[3];
```

What is the value of: ***(g + 1)**

```
            ⇩        ⇩
int g[ ] = {2, 4, 5, 8, 10, 32, 78};
            0  1  2  3   4   5   6  → positions in array

int *ptr1 = &g[0];
int *ptr2 = &g[3];
```

What is the value of: ***(g + 1)**

4 = answer.

Go to g, position zero.
Move over one address
Dereference and get the four.

```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};
            0  1  2  3   4   5   6 → positions in
                                       array

int *ptr1 = &g[0];
int *ptr2 = &g[3];
```

What is the value of: ***(g + 5)**

⬇ ⬇

int g[ ] = {2, 4, 5, 8, 10, 32, 78};

0 1 2 3 4 5 6 → positions in array

int *ptr1 = &g[0];

int *ptr2 = &g[3];

What is the value of: ***(g + 5)**

32 = answer

Go to g position zero
Move over 5 address
Dereference and get the 32

```
            ↓        ↓
int g[ ] = {2, 4, 5, 8, 10, 32, 78};
            0  1  2  3   4   5   6 -> positions in array

int *ptr1 = &g[0];
int *ptr2 = &g[3];
```

What is the value of: ***ptr1**

```
            ⬇        ⬇
int g[ ] = {2, 4, 5, 8, 10, 32, 78};
            0  1  2  3   4   5   6 -> positions in array

int *ptr1 = &g[0];
int *ptr2 = &g[3];
```

What is the value of: ***ptr1**

2 = answer

Find what ptr1 points to
Dereference and get the 2

```
            ⬇        ⬇
int g[ ] = {2, 4, 5, 8, 10, 32, 78};
            0  1  2  3  4   5   6 -> positions in array
```

```
int *ptr1 = &g[0];
int *ptr2 = &g[3];
```

What is the value of: ***ptr2**

```
            ⬇        ⬇
int g[ ] = {2, 4, 5, 8, 10, 32, 78};
            0  1  2  3   4    5    6 -> positions in array

int *ptr1 = &g[0];
int *ptr2 = &g[3];
```

What is the value of: ***ptr2**

8 = answer

Find what ptr2 points to
Dereference and get the 8

```
               ⬇        ⬇
int g[ ] = {2, 4, 5, 8, 10, 32, 78};
            0  1  2  3   4   5   6 -> positions in array
```

```
int *ptr1 = &g[0];
int *ptr2 = &g[3];
```

What is the value of: ***(ptr1 + 1)**

⇩ ⇩

int g[ ] = {2, 4, 5, 8, 10, 32, 78};

0 1 2 3 4 5 6 -> positions in array

int *ptr1 = &g[0];
int *ptr2 = &g[3];

What is the value of: ***(ptr1 + 1)**

4 = answer

Find what ptr1 points to (position zero)
Move over one address (position one)
Deference and get the 4

```
         ⬇     ⬇
int g[ ] = {2, 4, 5, 8, 10, 32, 78};
            0  1  2  3   4   5   6 -> positions in array
```

```
int *ptr1 = &g[0];
int *ptr2 = &g[3];
```

What is the value of: ***(ptr2 + 2)**

```
            ⬇        ⬇
int g[ ] = {2, 4, 5, 8, 10, 32, 78};
            0  1  2  3   4   5   6 -> positions in array

int *ptr1 = &g[0];
int *ptr2 = &g[3];
```

What is the value of: ***(ptr2 + 2)**

32 = answer

Find what ptr2 points to (position 3)
Move over 2 addresses (position 5)
Dereference and get the 32

```
            ⬇        ⬇
int g[ ] = {2, 4, 5, 8, 10, 32, 78};
            0  1  2  3   4   5   6 -> positions in array
```

```
int *ptr1 = &g[0];
int *ptr2 = &g[3];
```

What is the value of: ***ptr2 + 10**

```
                 ⬇           ⬇
int g[ ] = {2, 4, 5, 8, 10, 32, 78};
            0  1  2  3   4   5   6 -> positions in array

int *ptr1 = &g[0];
int *ptr2 = &g[3];
```

What is the value of: ***ptr2 + 10**

18 = answer

Find what ptr2 points to (position 3)
Dereference and get 8
Add 8 + 10 and get 18

Pointers and Functions

Pointers and Functions

Functions send arguments by *call-by-value*

The following exceptions use *call-by-address*:

Arrays – Address of array is passed to the function

Pointers – Address of variable, array, or string of characters is passed-to/returned-from a function

or

the pointer is used to step through an array

Example: a function to switch two values

```
void switch_it(int *a, int *b)
{
        int hold;
        hold = *a;
        *a = *b;
        *b = hold;
        return;
}
```

A valid call to this function would be:

```
int x, y;
switch_it(&x, &y);
```

Function Prototype is: **void switch_it(int *a, int *b);**

Below is a call to the switch_it function.
Is it a valid call?

```
float x = 1.5, y = 3.0, *ptr_x = &x, *ptr_y = &y;

switch_it(ptr_x, ptr_y);
```

Will NOT work since x & y are *float*, but the function requires the incoming arguments to be *int*

Function Prototype is: **void switch_it(int *a, int *b);**

Below is a call to the switch_it function.
Is it a valid call?

```
int f = 2, g = 7, *ptr_f = &f, *ptr_g = &g;

switch_it(ptr_f, ptr_g);
```

OK. All *int*. Passes in the addresses of f & g

Function Prototype is: **void switch_it(int *a, int *b);**

Below is a call to the switch_it function.
Is it a valid call?

```
int f = 2, g = 7, *ptr_f = &f, *ptr_g = &g;

switch_it(*ptr_f, *ptr_g);
```

No good! not passing the *address* of the f & g
but rather the *values* of 2 & 7

Function Prototype is: **void switch_it(int *a, int *b);**

Below is a call to the switch_it function.
Is it a valid call?

```
int f = 2, g = 7, *ptr_f = &f, *ptr_g = &g;

switch_it(&ptr_f, &ptr_g);
```

No good! Passing the addresses of the *pointers*
not the addresses of the *integers*.

Function Prototype is: **void switch_it(int *a, int *b);**

Below is a call to the switch_it function.
Is it a valid call?

```
int f = 2, g = 7, *ptr_f = &f, *ptr_g = &g;

switch_it(&f, &g);
```

OK. the addresses of f and g are being passed.

Function Prototype is: **void switch_it(int *a, int *b);**

Below is a call to the switch_it function.
Is it a valid call?

```
int f = 2, g = 7, *ptr_f = &f, *ptr_g = &g;

switch_it(f, g);
```

No good. This passing the *values* of f & g,
not the *addresses* of f & g.

Using the const Qualifier with Pointers

Using the **const** Qualifier with Pointers

- **const** – a keyword
- **const** qualifier
  - Variable cannot be changed
  - Use **const** if function does not need to change a variable
  - Attempting to change a **const** variable produces an error

Using the const Qualifier with Pointers. Examples.

int *const myPtr = &x;

*Type **int *const** – constant pointer to an **int***

ERROR:

int *const myPtr = &x;

myPtr = &b;

because we are trying to change the address.

The *const freezes the pointer.

Using the const Qualifier with Pointers. Examples.

const int *myPtr = &x;

Regular pointer to a **const int**

ERROR:

const int *myPtr = &x;

*myPtr = 9;

because we are not allowed to change the value

of x because the position of the * causes the value of x to freeze.

Using the const Qualifier with Pointers. Examples.

const int *const Ptr = &x;

const pointer to a **const int**

Nothing can be changed.

Function Pointers

What are function Pointers?

- C does not require that pointers only point to data, it is possible to have pointers to functions

- Functions occupy memory locations therefore every function has an address just like each variable

- Function pointers are different from regular pointers. They point to a function as opposed to a value. Hence they behave differently.

Why do we need function Pointers?

- Useful when alternative functions may be used to perform similar tasks on data (eg: sorting)
- One common use is in passing a function as a parameter in a function call.
- Can pass the data and the function to be used to some control function
- Greater flexibility and better code reuse

Define a Function Pointer

A function pointer is nothing else than a variable, it must be defined as usual.

```
int (*funcPointer) (int, char, int);
```

funcPointer is a pointer to a function.

The extra parentheses around (*funcPointer) is needed because there are precedence relationships in declaration just as there are in expressions

Assign an address to a Function Pointer

```
 //assign an address to the function pointer
int (*funcPointer) (int, char, int);

int firstExample ( int a, char b, int c) {
   printf(“ Welcome to the first example”);
   return a+b+c;
}
funcPointer= firstExample;      //assignment of address of
                                  the function to a pointer
funcPointer=&firstExample;    //alternative using
                              //address operator
```

Calling a function using a Function Pointer

There are two alternatives

1) Use the name of the function pointer
2) Can explicitly dereference it

```
int (*funcPointer) (int, char, int);
```

```
// calling a function using function pointer
        int answer= funcPointer (7, ’A’ , 2 );
        int answer=(* funcPointer) (7, ’A’ , 2 );
```

Example Trigonometric Functions

```
// prints tables showing the values of cos,sin
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
void tabulate(double (*f)(double), double first, double last, double incr);
int main(void) {
        double final, increment, initial;
        printf (“Enter initial value: “);
        scanf (“%lf”, &initial);
        printf (“Enter final value: “);
        scanf (%lf”, &final);
        printf (“Enter increment : “);
        scanf (%lf”, &increment);
        Printf(“\n    x   cos(x) \n”
             “  ----------  -----------\n”);
        tabulate(cos, initial,final,increment);
        Printf(“\n     x    sin (x) \n”
        “  ----------  -----------\n”);
        tabulate(sin, initial,final,increment);
        return (EXIT_SUCCESS);
}
```

The **main** function in little print. Bigger print used in following slides.

Example Trigonometric Functions (1 of 4)

```
// prints tables showing the values of cos, sin

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

void tabulate(double (*f)(double), double first,
                      double last, double incr);

int main(void)
```

Example Trigonometric Functions (2 of 4)

```
int main(void)
{
        double final, increment, initial;
          // Enter the data at the keyboard
        printf (“Enter initial value: “);
        scanf (“%lf”, &initial);
        printf (“Enter final value: “);
        scanf (%lf”, &final);
        printf (“Enter increment : “);
        scanf (%lf”, &increment);
```

Example Trigonometric Functions (3 of 4)

```
       // Print the headers and call tabulate
    printf(“\n    x   cos(x) \n”
        “  ----------  -----------\n”);
    tabulate(cos, initial,final,increment);

    printf(“\n     x    sin (x) \n”
    “  ----------  -----------\n”);
    tabulate(sin, initial,final,increment);
    return (EXIT_SUCCESS);
}
```

Trigonometric Functions (4 of 4)

```
// when passed a pointer f, the function prints a table
// showing the value of f

void tabulate(double (*f) (double), double first,
                        double last, double incr)
{
    double x;
    int i, num_intervals;
    num_intervals = ceil ( (last -first) /incr );
    for (i=0; i<=num_intervals; i++) {
       x= first +i * incr;
       printf(“%10.5f %10.5f\n”, x , (*f) (x));
     }
}
```

Output of the Example

```
Enter initial value: 0
Enter final value: .5
Enter increment: .1

     X            cos(x)
 ----------     -----------
 0.00000        1.00000
 0.10000        0.99500
 0.20000        0.98007
 0.30000        0.95534
 0.40000        0.92106
 0.50000        0.87758

    X             sin(x)
 ----------     -----------
 0.00000        0.00000
 0.10000        0.09983
 0.20000        0.19867
 0.30000        0.29552
 0.40000        0.38942
 0.50000        0.47943
```

Another Common Use of FuncPtr

- Sorting function (**qsort**)where you pass in a pointer to a comparison function that will return the results of the comparison.
  - Ex: Which argument was larger.

C-8 Pointers

The End