Pointers in C

Variable

- A variable is a named memory location.
- Variables provide direct access to its memory location.
- A variable has a name, an address, a type,and a value:
- "the name identifies the variable to the programmer
- "the address specifies where in main memory the variable is located

What is a variable?

- "the type specifies how to interpret the data stored in main memory and how long the variable is
- "the value is the actual data stored in the variable after if has been interpreted according to a given type

Pointer variable

- A pointer is a variable that contains the memory location of another variable.
- Syntax:-
- **type * variable name**
- You start by specifying the type of data stored in the location identified by the pointer.
- The asterisk tells the compiler that you are creating a pointer variable.
- Finally you give the name of the variable.

Declaring a Pointer Variable

- To declare ptr as an integer pointer:

```
int *ptr;
```

- To declare ptr as a character pointer:

```
char *ptr;
```

Address operator:

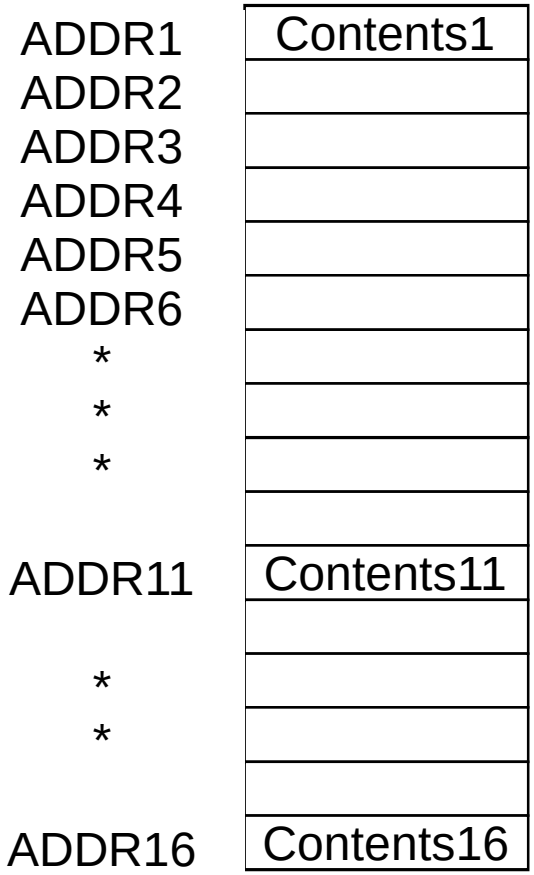
- Once we declare a pointer variable we must point it to something we can do this by assigning to the pointer the address of the variable you want to point as in the following example:

  ptr=#

- This places the address where num is stores into the variable ptr. If num is stored in memory 21260 address then the variable ptr has the value 21260.

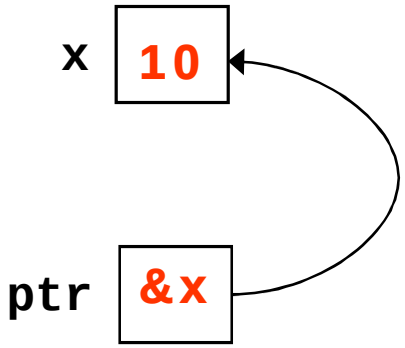
Address and Pointers

- Memory can be conceptualized as a linear set of data locations.
- Variables reference the contents of a locations
- Pointers have a value of the address of a given location



Pointer Variable

Assume **ptr** is a pointer variable and **x** is an integer variable




```
x = 10

ptr = &x
```

Now ptr can access the value of x.

HOW!!!!

Write: `*variable`. For example:

```
printf("%d\n", *ptr);
```

Can you tell me why we put %d????

Variables, Addresses and Pointers

```
main()
{
int a = 5;
int b = 6;
int* c;
// c points to a
c = &a;
}
```

Memory	Value
a (1001)	5
b (1003)	6
c (1005)	1001

```
include< stdio.h >
{
int num, *intptr;
float x, *floptr;
char ch, *cptr;
num=123;
x=12.34;
ch='a';
intptr=&num;
cptr=&ch;
floptr=&x;
printf("Num %d stored at address %u\n",*intptr,intptr);
printf("Value %f stored at address %u\n",*floptr,floptr);
printf("Character %c stored at address %u\n",*cptr,cptr);
}
```

Run this code

```
int main()
{
  int x;
  int *ptr;

  x = 10;
  ptr = &x;
  *ptr = *ptr + 1;
  printf(“x = %d\n”, x);
}
```

Manipulating Pointer Variable

- Once a variable is declared, we can get its address by preceding its name with the unary **&** operator, as in **&k**.
- We can "dereference" a pointer, i.e. refer to the value of that which it points to, by using the unary '*' operator as in ***ptr**

Reference

- Reference **&**
- **Retrieve the memory address of a variable**
- **int a = 6;**
- **int* c = &a; // &a is the memory location of variable a**

Dereference

- Dereference *** Accessing the variable (content) the pointer points to**
- (Indirection)
- **int a = 6;**
- **int* c = &a;**
- ***c = 7; /* Changes content of variable a by using its address stored in pointer c */**
- equivalent to
- **a = 7;**

From Program 9-9

```
 7     const int SIZE = 8;
 8     int set[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
 9     int *numPtr;   // Pointer
10     int count;     // Counter variable for loops
11
12     // Make numPtr point to the set array.
13     numPtr = set;
14
15     // Use the pointer to display the array contents.
16     cout << "The numbers in set are:\n";
17     for (count = 0; count < SIZE; count++)
18     {
19        cout << *numPtr << " ";
20        numPtr++;
21     }
22
23     // Display the array contents in reverse order.
24     cout << "\nThe numbers in set backward are:\n";
25     for (count = 0; count < SIZE; count++)
26     {
27        numPtr--;
28        cout << *numPtr << " ";
29     }
```

Program Output

```
The numbers in set are:
5 10 15 20 25 30 35 40
The numbers in set backward are:
40 35 30 25 20 15 10 5
```

Constant Pointers

- A constant pointer, **`ptr,`** is a pointer that is initialized with an address, and cannot point to anything else.
- We can use **`ptr`** to change the contents of **`value`**
- Example

```
int value = 22;
int * const ptr = &value;
```

Constant Pointer

- Constant pointer means the pointer is constant. Constant pointer is NOT pointer to constant.
- For eg:
  int * const ptr2 indicates that ptr2 is a pointer which is constant. This means that ptr2 cannot be made to point to another integer.
- However the integer pointed by ptr2 can be changed.

```
 //const pointer void
main()
 {
int i = 100,k;
int* const pi = &i;
*pi = 200;
pi=&k; //won't compile
}
```

Constant Pointers

```
int value = 22;
int value4 = 99;
int * const ptr = &value;
printf(“value:%d \t ”,value);
value=99;
printf(“value:%d \t ”,value);
ptr = &value4;
printf(“value4:%d \t ”,value4);
```

Constant Pointers to Constants

- A constant pointer to a constant is:
  - a pointer that points to a constant
  - a pointer that cannot point to anything except what it is pointing to

- Example:

```
int value = 22;
const int * const ptr = &value;
```

```
//const pointer to a const void
f3()
 {
 int i = 100;
 const int* const pi = &i;
 //*pi = 200; <- won't compile
 //pi++; <- won't compile
 }
```

Constant Pointer to a Constant

```
int value5 = 55;
int value12 = 1212;
const int * const ptr5 = &value5;
printf(“value:%d \t ”,value5);
value5=777;
printf(“value:%d \t ”,value);
  ptr5 = &value12;
  *ptr5=6543;
```

Pointer to constant

Pointer to constant is

const int * ptr1 indicates that ptr1 is a pointer that points to a constant integer. The integer is constant and cannot be changed. However, the pointer ptr1 can be made to point to some other integer.

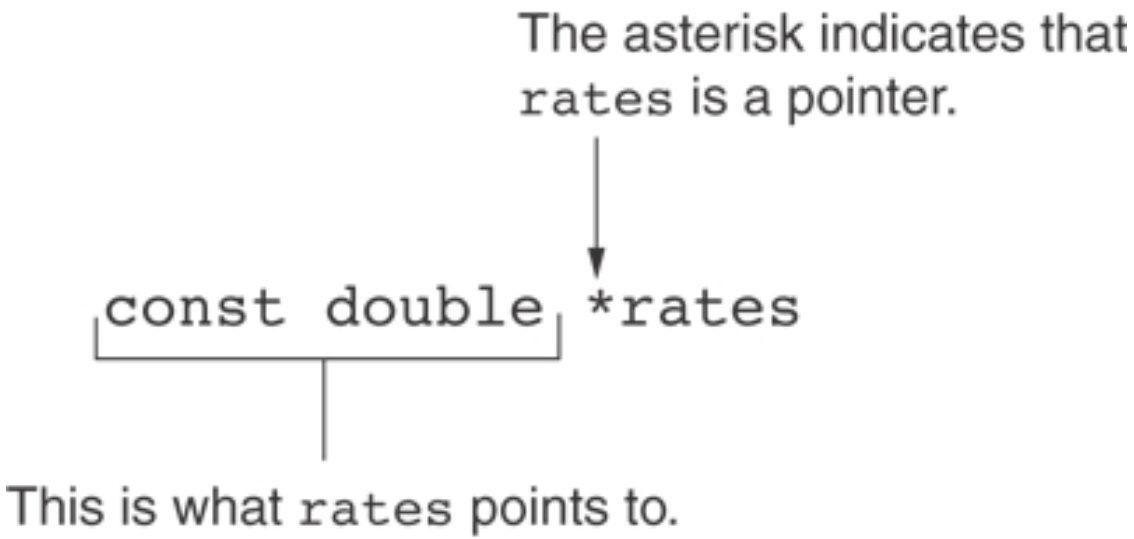
- //pointer to a const

```
void f1()
 {
int i = 100;
const int* pi = &i;
//*pi = 200; <- won't compile
pi++;
}m
```

Pointers to Constants

- To pass the address of a **`const`** item into a pointer, the pointer must be defined as a pointer to a **`const`** item
  - What is the purpose of a **`const`** item?

Pointer To a Constant



Pointer to a Constant

```
int value6 = 666;
int value7 = 777;
const int * ptr6 = &value6;
printf("value:%d \t ",value6);
printf("value:%d \t %d ",value6,*ptr6);
value6=6666;
printf("value:%d \t %d ",value6,*ptr6);
ptr6=&value7;
printf("value:%d \t %d ",value7,*ptr6);
*ptr6=6543;
```

Pointer arithmetic

Valid operations on pointers include:

- the *sum of a pointer and an integer*
- the *difference of a pointer and an integer*
- pointer *comparison*
-the *difference of two pointers.*
-*Increment/decrement in pointers*
- *assignment operator used in pointers*

Example

```
 void main()
{
 int a=25,b=78,sum;
 int *x,*y;
 x=&a;
 y=&b;
  sum= *x + *y;
 printf(“Sum is : %d”,sum);
}
```

- ***p = 1;**
- ***(p + 1) = 2;**
- ***(p + 2) = 3;**

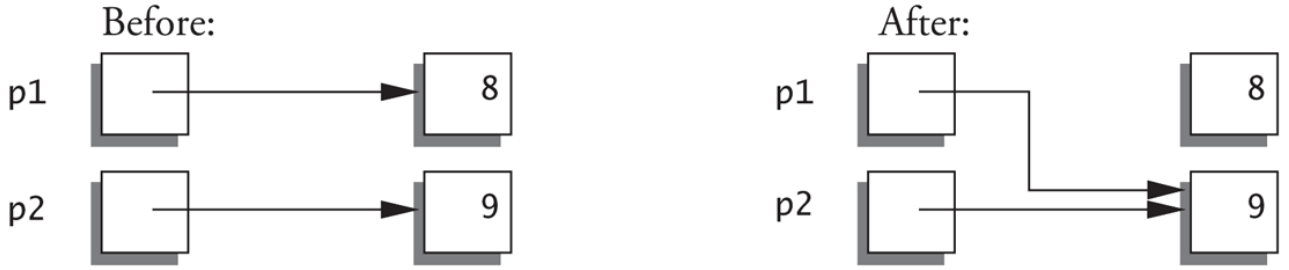
Assignment in pointers

- Pointer variables can be "assigned":
  int *p1, *p2;
  p2 = p1;
  - Assigns one pointer to another
  - "Make p2 point to where p1 points"
- Do not confuse with:
  *p1 = *p2;
  - Assigns "value pointed to" by p1, to "value pointed to" by p2

Diagrammatic representation

Display 10.1 Uses of the Assignment Operator with Pointer Variables

```
p1 = p2;
```




```
*p1 = *p2;
```

Comparison in pointers

- Two pointers of the same type, **p and q, may be compared as long**
- as *both of them point to objects within a single memory block*
- • Pointers may be compared using the <, >, <=, >=, == , !=
- • When you are comparing two pointers, you are comparing the
- values of those pointers *rather than the contents of memory* locations pointed to by these pointers

Pointer Arithmetic

```
int vals[]={4,7,11};
int *valptr = vals;
printf("\nvalue of valptr++: “,valptr++);
printf("\nvalue of *valptr: “,*valptr);

printf("\nvalue of valptr--: “,valptr--);
printf("\nvalue of *valptr: “,*valptr);

printf(“value of *(valptr+2): “, *(valptr + 2));

valptr = vals;
valptr+=2;
 printf(“value of valptr+=2: “,valptr);
```

- You can perform a limited number of arithmetic operations on pointers. These operations are:
- **Increment and decrement**
- **Addition and subtraction**
- **Comparison**
- **Assignment**
- The increment (++) operator increases the value of a pointer by the size of the data object the pointer refers to. For example, if the pointer refers to the second element in an array, the ++ makes the pointer refer to the third element in the array.
- The decrement (--) operator decreases the value of a pointer by the size of the data object the pointer refers to. For example, if the pointer refers to the second element in an array, the -- makes the pointer refer to the first element in the array.

- You can add an integer to a pointer but you cannot add a pointer to a pointer.
- If the pointer p points to the first element in an array, the following expression causes the pointer to point to the third element in the same array:
- p = p + 2; If you have two pointers that point to the same array, you can subtract one pointer from the other. This operation yields the number of elements in the array that separate the two addresses that the pointers refer to.
- You can compare two pointers with the following operators: ==, !=, <, >, <=, and >=.
- Pointer comparisons are defined only when the pointers point to elements of the same array. Pointer comparisons using the == and != operators can be performed even when the pointers point to elements of different arrays.
- You can assign to a pointer the address of a data object, the value of another compatible pointer or the NULL pointer.

Pointer Arithmetic

- Operations on pointer variables:

Operation	**Example**
	`int vals[]={4,7,11};`
	`int *valptr = vals;`

`++, --`	`valptr++; // points at 7`
	`valptr--; // now points at 4`

`+, -` (pointer and `int`)	`cout << *(valptr + 2); // 11`

`+=,-=` (pointer and `int`)	`valptr = vals; // points at 4`
	`valptr += 2;  // points at 11`

`-` (pointer from pointer)	`cout << valptr–val; // difference`
	`//(number of ints) between valptr`
	`// and val`

Generic pointers

- When a variable is declared as being a pointer to type void it is known as a generic pointer.
- Since you cannot have a variable of type void, the pointer will not point to any data and therefore cannot be dereferenced.
- It is still a pointer though, to use it you just have to typecast it to another kind of pointer first. Hence the term Generic pointer.

- This is very useful when you want a pointer to point to data of different types at different times.

- Syntax:

  void * variable name;

  Print value stored in variable

  ***(data_type*)name of variable;**

```
void main()
{ int i;
char c;
void *the_data;
i = 6;
c = 'a';
the_data = &i;
printf("the_data points to the integer value %d\n",
*(int*) the_data);
the_data = &c;
printf("the_data now points to the character %c\n",
*(char*) the_data);
return ;
}
```

Null Pointer

- NULL value can be assigned to any pointer, no matter what its type.
- void *p = NULL;
- int i = 2;
- int *ip = &i;
- p = ip;
- printf("%d", *p);
- printf("%d", *((int*)p));