Programming Fundamentals for Engineers - 0702113

8. Pointers

YACOUB SABATIN
MUNTASER ABULAFI
OMAR QARAEEN

1

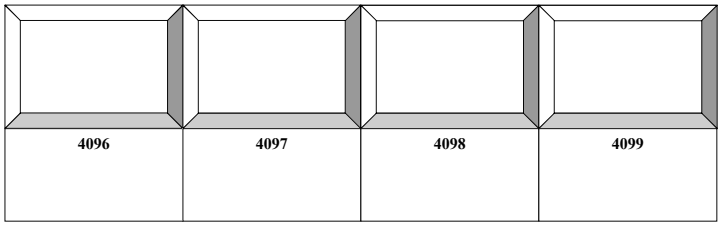
- A **pointer:**
  - Powerful, but difficult to master,
  - Is a derived data type,
  - Built from **one** of the standard types,
  - Contains **the address** of the **first byte** of a memory variable, ***Regardless*** of the type of the memory variable **and** the amount of memory that it occupies,
  - Can only be used to reference the first variable; i.e. Simulate **call-by-reference**
- Especially valuable when working with strings. There is an **intimate link** between **arrays** and **pointers** in C.

2

- **C** uses **pointers** in three different ways **to**:
  - **a. Create Dynamic Data Structures➔ DS** is built up from **blocks** of memory allocated from the heap at run-time.
    - **Heap** is a specialized **tree-based DS** that satisfies the ***heap*** *property:*
  - **if *B* is a child node of *A*, then key(*A*) ≥ key(*B*)**
  - **b. Handle** variable parameters passed to functions,
  - **c. Provide** an alternative way **to access** information stored in **arrays**.
- **Recalling**:
  - **Declaring a variable ➔ Reserving** bytes in the computer's memory for that variable;

3

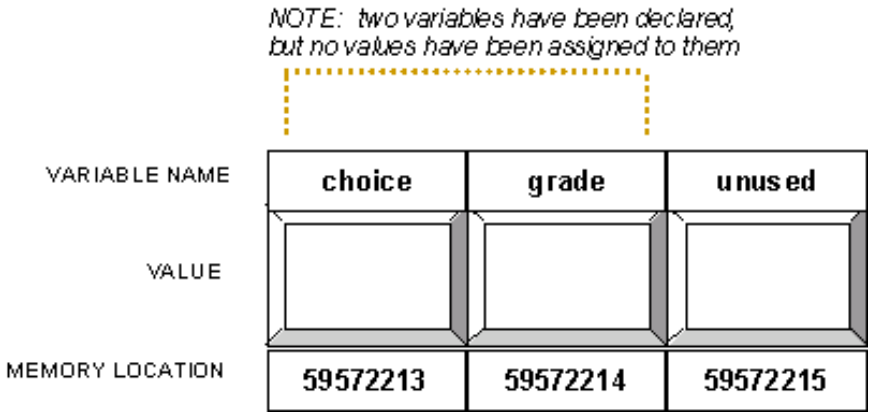
  - Computer memory is divided into **Bytes;**
  - Each has an **address** expressed in **hex** (sequentially numbered locations)
- Ex: An integer variable **i** resides at address 4096➔



- You can picture the computer's memory as a very long **row of slots**
- *It's like a long road of* ***houses*** - each **house** has a **unique number** in decimal format. In each occupied **house**, there are **people**;

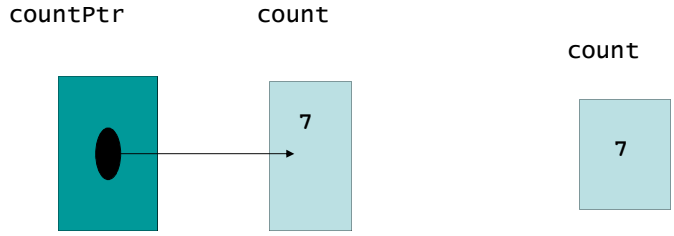
4

- **Houses are memory slots,**
- **People are the variables/data.**
- For example: **char choice;**
  char grade;



5

- **Pointer variables:**
  - Contain memory **addresses** as their **values**
  - Normal variables contain a specific value (**direct reference**)



  - **Pointer** contains **address of a variable** that has a **specific value** (**indirect reference**)
  - **Indirection ➔ Referencing a pointer value;**

6

The & Operator

- Gives the memory address of an object

```
char c = 'A';
```

c:

`&c` yields the value 0x2000

- Also known as the "address operator."

7

Example:

```
char c;
printf("The address of c: %p \n", &c);
```

"conversion specifier" for printing a memory address

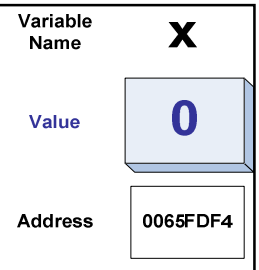
8

So…

- A pointer is a variable.
- Points to another variable.
- Contains a memory address.
- Points to a specific data type.
- As a convention, pointer variables are usually named ***varPtr*** or ***p_var***, where ***var*** is the name of the variable this pointer points to.

9

```
#include <stdio.h>
main()
{
  int x = 0;
  printf("Address of x = 0x%p \n", &x);
  return 0;
}
```



Output: Address of x = 0x0065FDF4

→ **%p** is the format specifier for **addresses**

→ **0x in the printf function signifies** that a **hex number follows**

10

Declaring a Pointer

- Suppose we had an **integer variable**, **x**:
  - We know how to get the **address** of **x** using **&**, BUT
  - How to store the **hex** value returned by **&x**?
- This is where ***pointers*** come into play!

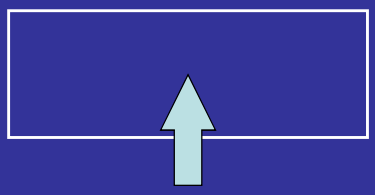
11

Example:

```
char* cPtr;
```

cPtr:

0x2000



Can store an address of variables of type **`char`**

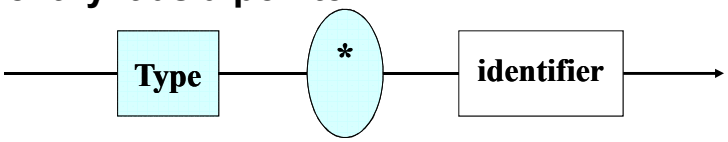
- We say cPtr is a pointer to char.

12

– A **pointer to a variable** named **choice** would be named **p_choice;**

– **A pointer variable is *always* declared:**

- **To be the same type as the variable it points to;** and
- **Must have a * (an asterisk) in front of it, to identify it as a pointer.**



- **E.g. 1**:
  – Declare a pointer to **point to a char** variable named **initial**, the declaration would be:

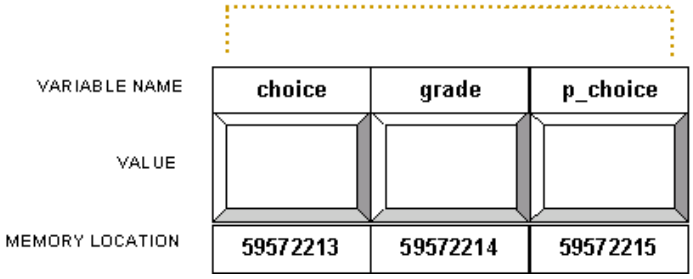
char *p_initial; /* pointer to a char */

13

– Consider the following declarations:

char choice;
char grade;
char *p_choice;

AGAIN: So far we have only declared the two char variables and the pointer. The pointer has not been set to point to anything yet and the other two char variables have no values. Results would be very unpredictable if you tried to print or use an uninitialized pointer. Remember declaring is not the same thing as initializing.



14

→**int *p;** → **p** is a pointer variable that can point to a variable of type integer

→**p = &i;** → Assigns address of **i** to pointer variable **p**

→**int *p = &i;** → Initializes pointer at time of declaration (**i** must be previously declared)

→**int i, *p = &i;** → Combine declaration of **i** and **p**, (**i** must come first)

- Don't confuse this with the BINARY multiplication operator → Multiply takes two operands (x*y),
- The **data type of a pointer *must* match the data type of the variable it points to**

15

Notes on Pointers

- We can have pointers to any data type.

Example:

```
int*    numPtr;
float*  xPtr;
```

- You can print the address stored in a pointer using the `%p` conversion specifier.

Example:

```
printf("%p", numPtr);
```

16

Notes on Pointers (cont.)

- The * can be anywhere between the type and the variable.

Example:

```
int    *numPtr;
float * xPtr;
```

- Each pointer must be declared with an *.

Example:

```
int   *aPtr, bPtr, *cPtr;
```

aPtr *and* cPtr *are integer pointers, but not* bPtr.

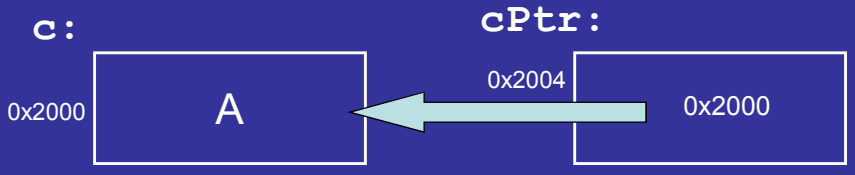
17

Pointers and the *&* Operator

Example:

```
char c = 'A';
char *cPtr;

cPtr = &c;
```

Assigns the address of c *to* cPtr.



18

The * Operator

- Allows pointers to access values stored in variables they point to.
- Also known as "de-referencing operator."
- Should not be confused with the * in the pointer declaration.

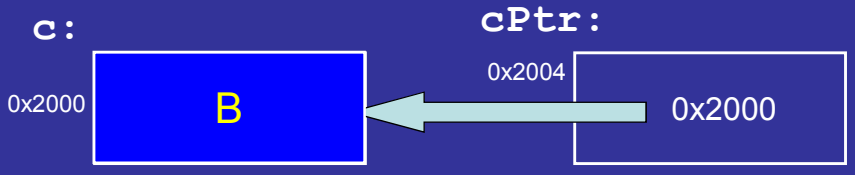
19

Pointers and the * Operator

Example:

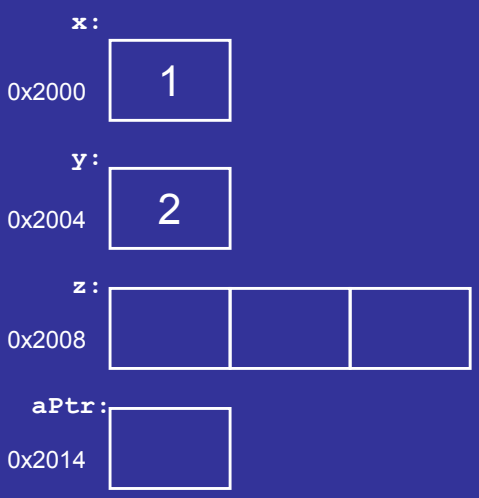
```
char c = 'A';
char *cPtr;

cPtr = &c;
*cPtr = 'B';
```

Changes the value of the variable which cPtr points to.



20

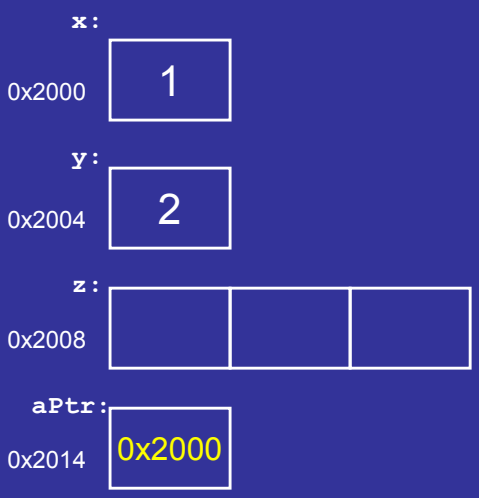
```
int   x = 1,  y =
2,  z[3];
int* aPtr;
```



Notice that these addresses are expressed in HEX.
Suppose `int` takes 4 bytes (0x2000 then 0x2004 ..etc)

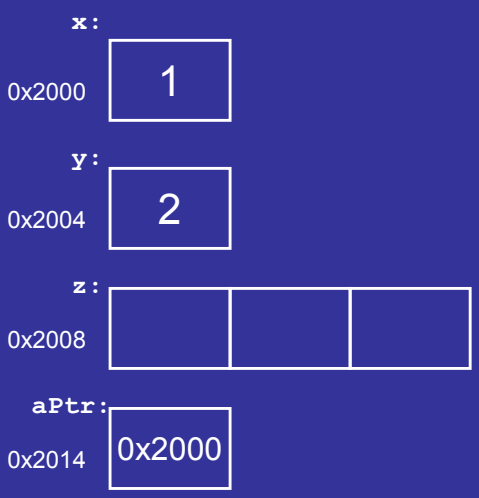
21

```
int   x = 1,  y =
2,  z[3];
int* aPtr;

aPtr = &x;
```



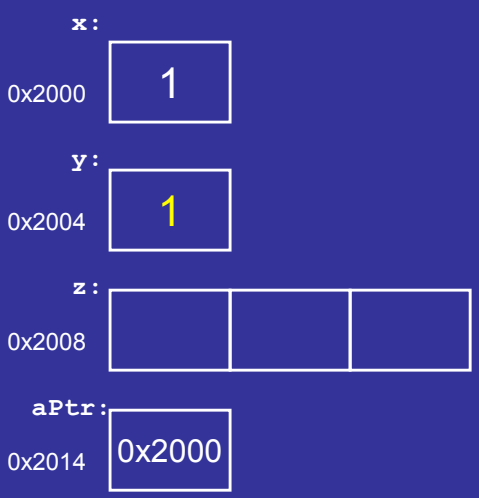
22

```
int   x = 1,  y =
2,  z[3];
int* aPtr;

aPtr = &x;
y = *aPtr;
```



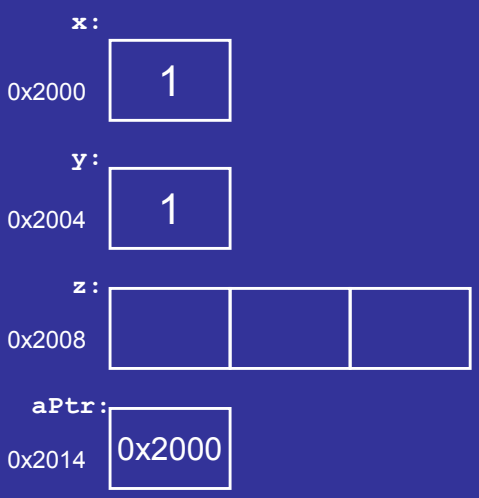
23

```
int   x = 1,  y =
2,  z[3];
int* aPtr;

aPtr = &x;
y = *aPtr;
```



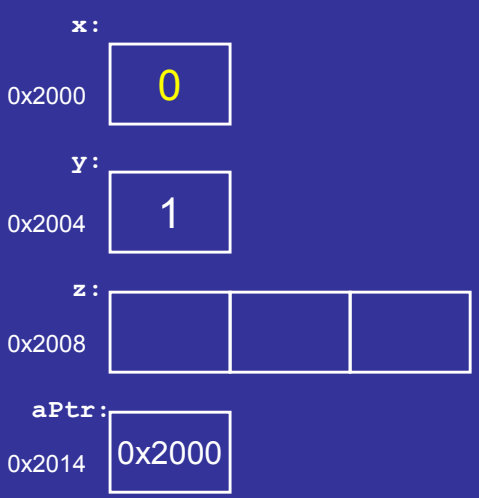
24

```
int   x = 1,  y =
2,  z[3];
int* aPtr;

aPtr = &x;
y = *aPtr;
*aPtr = 0;
```



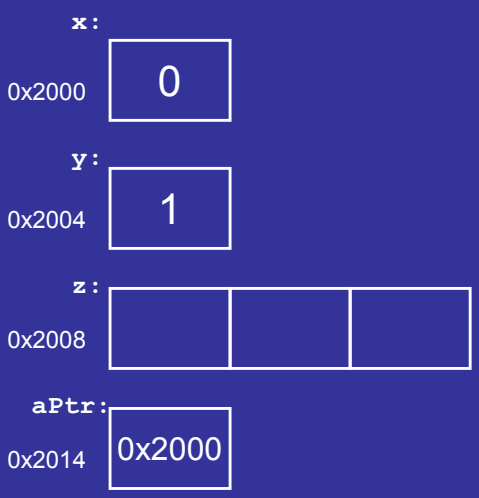
25

```
int   x = 1,  y =
2,  z[3];
int* aPtr;

aPtr = &x;
y = *aPtr;
*aPtr = 0;
```



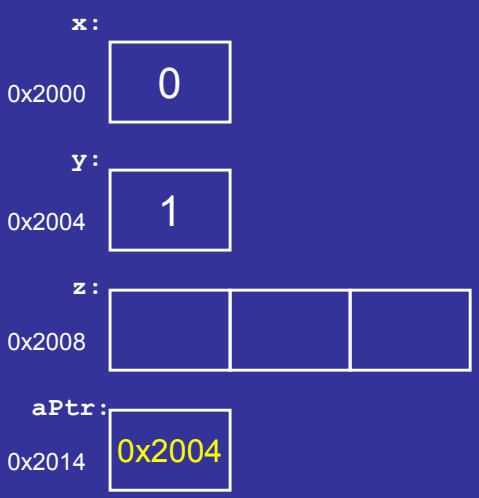
26

```
int   x = 1,  y =
2,  z[3];
int* aPtr;

aPtr = &x;
y = *aPtr;
*aPtr = 0;
aPtr = &y;
```



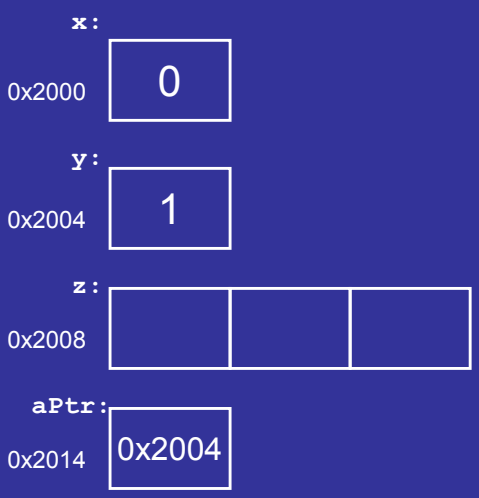
27

```
int   x = 1,  y =
2,  z[3];
int* aPtr;

aPtr = &x;
y = *aPtr;
*aPtr = 0;
aPtr = &y;
```



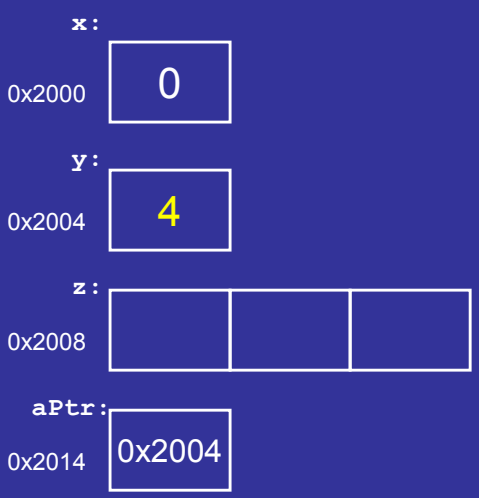
28

```
int   x = 1,  y =
2,  z[3];
int* aPtr;

aPtr = &x;
y = *aPtr;
*aPtr = 0;
aPtr = &y;
*aPtr = 4;
```



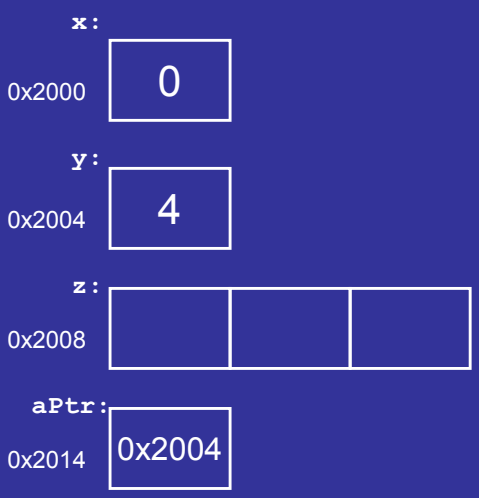
29

```
int   x = 1,  y =
2,  z[3];
int* aPtr;

aPtr = &x;
y = *aPtr;
*aPtr = 0;
aPtr = &y;
*aPtr = 4;
```



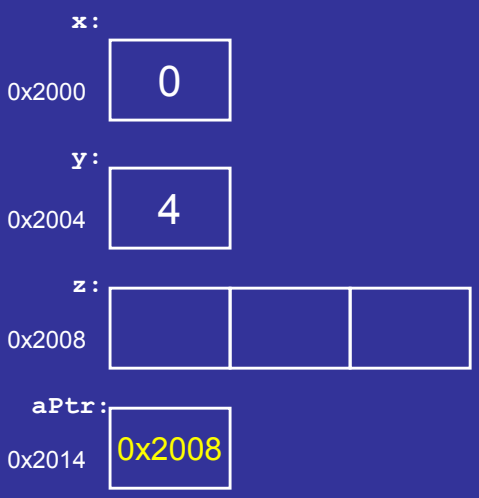
30

```
int   x = 1,  y =
2,  z[3];
int* aPtr;

aPtr = &x;
y = *aPtr;
*aPtr = 0;
aPtr = &y;
*aPtr = 4;
aPtr = &z[0];
```



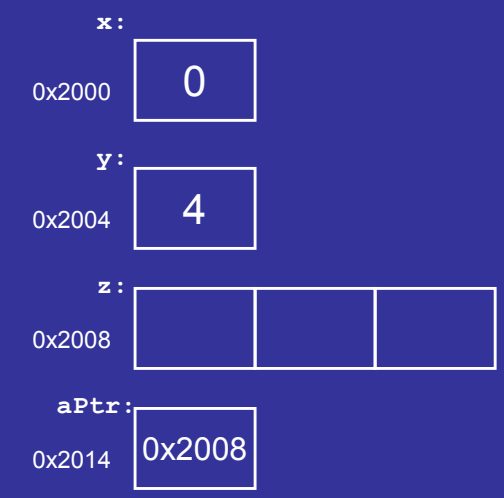
31

```
int   x = 1,  y =
2,  z[3];
int* aPtr;

aPtr = &x;
y = *aPtr;
*aPtr = 0;
aPtr = &y;
*aPtr = 4;
aPtr = &z[0];
```



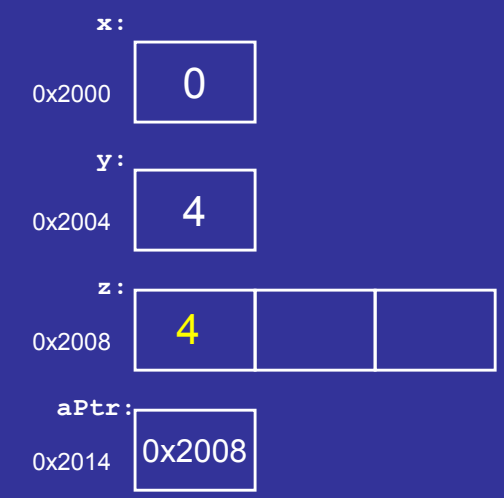
32

```
int   x = 1,  y =
2,  z[3];
int* aPtr;

aPtr = &x;
y = *aPtr;
*aPtr = 0;
aPtr = &y;
*aPtr = 4;
aPtr = &z[0];
*aPtr = y;
```



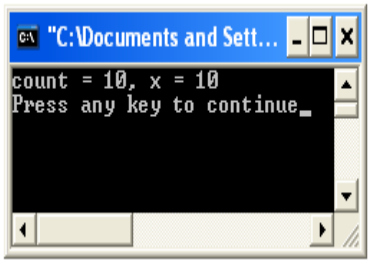
33

```
int   x = 1,  y =
2,  z[3];
int* aPtr;

aPtr = &x;
y = *aPtr;
*aPtr = 0;
aPtr = &y;
*aPtr = 4;
aPtr = &z[0];
*aPtr = y;
```



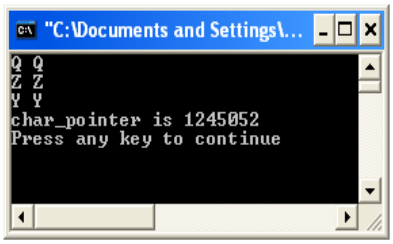
34

```c
#include <stdio.h>

    main()
    {
        int count  = 10, x, *int_pointer;

        /* this assigns the memory address of count to int_pointer  */
        int_pointer = &count;

        /* assigns the value stored at the address specified by int_pointer to x */
        x = *int_pointer;

        printf("count = %d, x = %d\n", count, x);

  return 0;
}
```



35

```c
#include <stdio.h>
   main()
   {
       char c = 'Q';
       char *char_pointer = &c;

       printf("%c %c\n", c, *char_pointer);

       c = 'Z';

       printf("%c %c\n", c, *char_pointer);

       *char_pointer = 'Y';
       /* assigns Y as the contents of the memory address specified by char_pointer  */

       printf("%c %c\n", c, *char_pointer);

       printf("char_pointer is %d\n", char_pointer);

       return 0;
   }
```



36

Pointer Operators:

- So, there are **two** special **pointer operators: *** and **&**
  - The **&** (**address-of**) operator ➔ Returns the **address of** the variable it precedes.
  - The ***** (**indirection**) operator ➔ Returns the **value** stored at the address that it precedes. This is also known as **dereferencing** the pointer.

Indirection Operator:

***char_pointer** and **c** both refer to the contents of **c**:
printf("The value of c is %c\n", c); and
printf("Value of char_pointer is %c\n", *char_poniter);

- Both statements are equivalent in results, and will print the **value** stored in **c**.

37

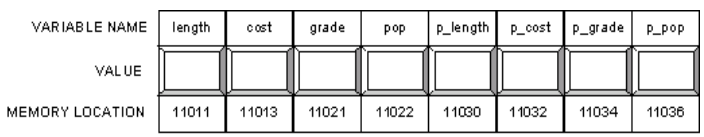
Assume we have the following four variables:

```
short int length ;
double cost ;
char grade ;
long pop ;
```

/* The appropriate data type for each of the four pointers will be: */

```
short int *p_length ; /* pointer to length */
double *p_cost ;      /* pointer to cost */
char *p_grade ;       /* pointer to grade */
long *p_pop;          /* pointer to pop */
```

The Memory Map:



38

Assigning value to Memory Block pointed to by pointer

We assign a value to what the pointer is pointing to by placing a * (star) operator in front of the pointer variable.

p = 5; / assign the value 5 to the address represented by p */

address	name	contents
2000	p	3000

--->

address	value
3000	5

To assign a value to the memory location pointed to by a pointer p it's a 4 step process.

steps	assigning a value *p = 5
(1)	go to pointer at address 2000
(2)	get contents of pointer p (location 3000)
(3)	go to address location 3000 specified in pointer
(4)	assign the value 5 to that address location

p

address	contents
2000	3000
3000	5

39

- Values of pointer variables can be assigned and/or copied provided they have **same** data type

```
int i, j, *p, *q;
p=&i;  // pointer assignment, address i copied into p
q = p; /* copies contents of p (i address) to q
          Making q point to same place as p
          So can change i by assigning a value to *p
          or *q  */
*p=1;
*q=2;
```

40

Reading Value Stored in Memory

We get a value pointed to by the pointer by using the * (star) operator placed in front of the pointer variable.

```
x = *p;    /* read value pointed to by p and assign to x */
```

address	name	contents
2000	p	3000

--->

address	value
3000	5

Assume variable x is at address location 1000.

address	name	value
1000	x	5

```
/* x will now have the value 5 */
```

What is the value of x ?

To read a value from the memory location pointed to by a pointer p it's a 5 step process.

steps	reading a value x = *p;
(1)	go to pointer at address 2000
(2)	get contents of pointer (location 3000)
(3)	go to address location (3000) specified in pointer
(4)	get the value at that address location (5)
(5)	assign the value 5 to the variable on the left hand side of the assignment statement

	address	contents
x	1000	5
p	2000	3000
	3000	5

Getting Address of Existing Variable

Declare two variables x and y, assign 5 to x;

```
int x = 5;    /* declare int variable x with initial value 5 */
int y;        /* declare int variable y  */
```

address	name	data value
1000	x	5
1002	y	??

To get the address of an existing variable we use the & (ampersand) operator

```
pointer = & variable name;
```

```
p = &x;    /* assign address of x to p */
```

address	name	contents
2000	p	&x (1000)

-->

address	name	value
1000	x	5

p now points to x and contains the address 1000.

What is the contents of p ? What is the value that p points to? What is the address of x ? What is the value of x ?

The contents of p is now the address of x. p points to x. If you change x to 3 then the memory contents that p points also has the value of 3. Why ? because p points to x.

```
x = 3;    /* assign 3 to x */
```

address	name	contents
2000	p	&x (1000)

-->

address	name	value
1000	x	3

What is p ? What is *p ? What is &x ? What x ?

42

What is p ? What is *p ? What is &x ? What x ?

Assign to y the value pointed to by p.

```
y = *p;        /* assign the contents of what p points to variable y */
```

address	name	value
1002	y	3

What is the value of y ? What is the value of x ?

If you assign a value to what p points to:

```
*p = 7,    /* assign 7 to what p points to */
```

address	name	contents
2000	p	&x (1000)

-->

address	name	value
1000	x	7

What is x, p, *p, y ?

What is the difference between the address of p and the contents of p and the value pointed to by p ? A pointer is located at an address in memory and contains an address to a memory location.

& means address of
* means the value pointed to by the pointer

43

*Recalling: There are only **three** basic ideas:*

1. To **declare** a **pointer** **add** an * in front of its name.
2. To **obtain** the **address** of a **variable** **use** & in front of its name.
3. To **obtain** the **value** of a **variable** **use** * in front of its pointer's name.

Note: * and & are inverses

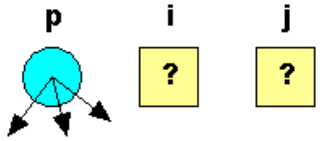
```
int *a , b , c; // a is a pointer to int, b and c are integers
b = 10;      // Assigning 10 to b
a = &b;      // Assigning address of b to a ➔ a points to b
c = *a;      /* Stores the value in the variable pointed to
                by a in c.  As a points to b, its value i.e. 10 is
                stored in c. That is c = b; */
```

44

```c
#include <stdio.h>
int main()
{
   int i,j;      /* declare two normal integers */
   int *p;       /* a pointer to an integer */
   p = &i; // assign to p the address of i , p points to i
   *p=5;
   j=i;
   printf("%d %d %d\n", i, j, *p);
   return 0;
}
```

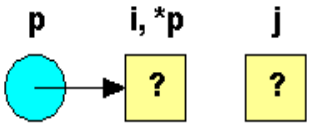
45

To visualize what is happening:



- 3 variables **i**, **j** and **p**
- **i** and **j** integer variables are drawn as boxes containing question marks (could contain any value at this point in the program's execution
- The **pointer** is drawn as a **circle** to distinguish it from a normal variable that holds a value, and the random arrows indicate that it can be pointing anywhere at this moment

46



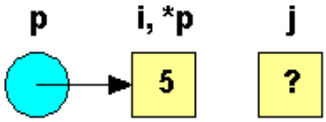
- **p = &i; ➔ p is initialized and it points to i**
- Once **p** points to **i**, the memory location **i** has <u>two names</u>. It is still known as **i**, but now it is known as ***p** as well. This is how C talks about the two parts of a pointer variable:

  p is the location holding the address,

while

  ***p is the location pointed to by that address**

47

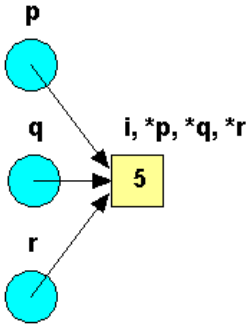


- ***p=5;**

  means that the location pointed to by **p** should be set to **5**
- Because the location ***p** is also **i**, **i** also takes on the value **5**.
- Consequently, **j=i;** sets **j to 5**, and
- ***printf*** statement produces 5 5 5

48

Any number of pointers can point to the same address




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

- Variable **i** now has four names: **i, *p, *q** and ***r**. There is **no limit** on the number of pointers that can hold (and therefore **point to**) the same address

49

Another Example:

```
#include <stdio.h>
int main()
{
  int x = 12;
  int *ptr = &x;

  printf("Address of x: 0x%p\n", ptr);
  printf("Address of x: 0x%x\n", &x);
  printf("Address of ptr: 0x%x\n", &ptr);
  printf("Value of x: %d\n", *ptr);

  return 0;
}
```

Output

```
Address of x: 0x0065FDF4
Address of x: 0x65fdf4
Address of ptr: 0x65fdf0
Value of x: 12
```

50

- The first **2 printf** statements demonstrate that the address of **x** is stored in the pointer variable, **ptr**. Notice how **%p** returns a full 8 digit hex value in uppercase - maybe **i** should've used **%x** for the address of **x** for clarity.
- The **3rd printf** demonstrates that **ptr** itself has a place in the memory, which shows it's a variable.
- The final **printf** uses the dereference operator to extract the value pointed to by **ptr**, like this: ***ptr**.

51

To demonstrate the use of pointers:

```
#include <stdio.h>
int main (void)
{
      int a;
      int *p;
      a=14;
      p=&a;
      printf(" %d %p\n" , a, &a);
      printf(" %d %p\n" , *p,p);
       (*p)++;
      printf(" %d %d\n",*p,a);
      p++;
      printf(" %d %p\n",*p,p);
return 0;
}
```

Declaration of a pointer variable

Initialization of a pointer variable

Output is:
14 2AE7:202E
14 2AE7:202E
15 15
10983 2AE7:2030

Content of a

Content of p

Format specifier for printing pointer values in hexadecimal notation.

52

1) Determine the output of the following code:

```
#include <stdio.h>
main()
{
      int count  = 10, x, *int_pointer;
      /* Assigns memory address of count to
      int_pointer */
      int_pointer = &count;
      /* Assigns the value stored at the address
      specified  by int_pointer to x */
      x = *int_pointer;
      printf("count = %d, x = %d\n", count, x);
      Return 0;
}
```

53

2) Determine the output of the following code:

```
#include <stdio.h>
main()
{
      char c = 'Q';
      char *char_pointer = &c;
      printf("%c %c\n", c, *char_pointer);
      c = '/';
      printf("%c %c\n", c, *char_pointer);
      *char_pointer = '(';
      printf("%c %c\n", c, *char_pointer);
      return 0;
}
```

54

3) Determine the output of the following code:

```
#include <stdio.h>
main()
{
 int i1, i2, *p1, *p2;
 i1 = 5;
 p1 = &i1;
 i2 = *p1 / 2 + 10;
 p2 = p1;
 printf("i1 = %d, i2 = %d, *p1 = %d, *p2
 = %d\n", i1, i2, *p1, *p2);
 return 0;
}
```

55

Answers:

1. count = 10, x = 10

2. Q Q

/ /

((

3. i1 = 5, i2 = 12, *p1 = 5, *p2 = 5

56

```c
/* Using the & and * operators */
#include <stdio.h>
int main()
{ int a;      // a is an integer
  int *aPtr;  // aPtr is a pointer to an integer
  a = 7;
  aPtr = &a;  // aPtr set to address of a
  printf( "The address of a is %p\nThe value of
           aPtr is %p", &a, aPtr );
  printf( "\n\nThe value of a is %d\nThe value of
                *aPtr is %d", a, *aPtr );
  printf( "\n\nShowing that * and & are
     complements of each other\n&*aPtr =
     %p\n*&aPtr = %p\n", &*aPtr, *&aPtr );
   return 0;
}
```

57

```
The address of a is 0012FF7C
The value of aPtr is 0012FF7C

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of
each other.
&*aPtr = 0012FF7C
*&aPtr = 0012FF7C
```

58

Additional Operators:

Operators								Associativity	Type
()	[]							left to right	highest
+	-	++	--	!	*	&	(type)	right to left	unary
*	/	%						left to right	multiplicative
+	-							left to right	additive
<	<=	>	>=					left to right	relational
==	!=							left to right	equality
&&								left to right	logical and
||								left to right	logical or
?:								right to left	conditional
=	+=	-=	*=	/=	%=			right to left	assignment
,								left to right	comma

Operator precedence.

59

- **Pass by Value vs. Pass by Reference:** We can either send:
  - A **copy** of the **variable** **(pass by value)** to a **function**; **OR**
  - The **address** of the **variable** to the **function** **(pass by reference)**.
- **Pass by reference:**
  - **Giving a function the key to the variable**.
  - The advantage is that the **function can use this key** (the **address**) to **store** the **results** in the **original variable's memory location**.

60

- We can send as many **addresses** to the **function** as we wish;
- Then our **function** **can calculate several results** and **pass** them all back to the **calling function** as needed, simply by:
  - **Storing the results at the memory location of the original variables.**

- **This is the idea behind using pointers:**
  - **Passing the address of the variable to the pointer.**

61

Function Calls & Function Argument

- One of the most useful **applications** of **pointers** are for **functions.**
- **Recall** that the function can **only return one value** as their **return arguments;**
- **In case,** we were to **return more than one value**, we had to pass these values by **reference**.
- **Variables** can be passed to a **function** (as **function arguments** when called) either:
  - By **Value**: *(as a **copy** of a **local variable**); OR*
  - By **Reference**: *(by a **pointer**).*

62

- **Function Argument: *Passing by Value***
  - This is the method we have used so far in all our examples, where a (local) **copy** of the **variable** is **made** & **passed** to the **function**;
  - **Changing** the (**passed**) **variable** within the **calling function** **has no effect** on the **original variable** that was passed.

63

```
void double_increment (int n)
{ n++; n++; }
main ()
{
   int n = 3;
   /* an attempt is made to change the
   value of n in main by passing it to
   double_increment */
   double_increment (n);
   /* n in main will remain unchanged as
   it is completely a separate variable
   from the n in double_increment   */
   printf ("n is %d\n",  n);
   return 0;
}
```

64

- **Function Argument:** ***Passing by Reference***
  - Allows us **to change** the value of a **variable** (**not Local to the function**) **without** having to make it **global;**
  - Passing a reference, **pointer**, to **function** that tells the function where **to find** that variable;

Concluding:

- Usually, **C** passes arguments to functions **by value;**
- Pointers can be passed as arguments to functions;
- ***The advantage:***
  If a **pointer to a variable** is **passed** to a **function**, **Then** that **function** can modify that **variable**;

65

- There are many cases when we want to **alter** a **passed argument** in the **function** and **receive** the **new value back once the function has finished**: ***C uses pointers explicitly to do this***
- Pointers provide the solution:

  Pass the address of the variables to the functions and access address of function;
- *** operator**: Used as a nickname for variable inside of function:

```
void double( int *number )
{  *number = 2 * ( *number ); }
```

`*number`: Used as nickname for the variable passed.

66

Pointers and Function Arguments

- Example: swap the values of two variables

67

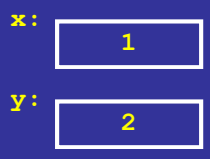
```
#include <stdio.h>

void
swap1(int a, int b)
{
   int tmp;

   tmp = a;
   a = b;
   b = tmp;
}

main()
{
   int x = 1, y = 2;

   swap1(x, y);
   printf("%d %d\n", x, y);
}
```

Solution 1

68

Solution 1

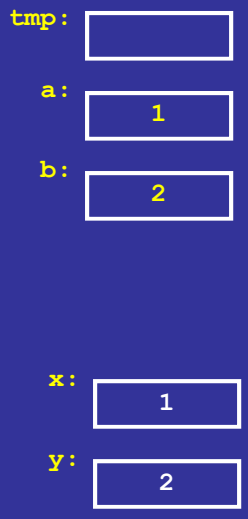
```
#include <stdio.h>

void
swap1(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
}

main()
{
    int x = 1, y = 2;

    swap1(x, y);
    printf("%d %d\n", x, y);
}
```



69

Solution 1

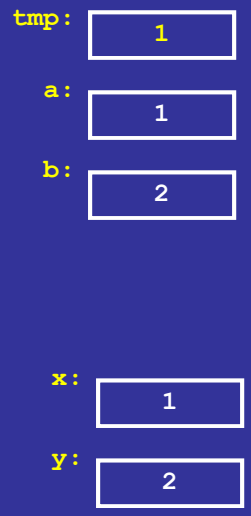
```
#include <stdio.h>

void
swap1(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
}

main()
{
    int x = 1, y = 2;

    swap1(x, y);
    printf("%d %d\n", x, y);
}
```



70

Solution 1

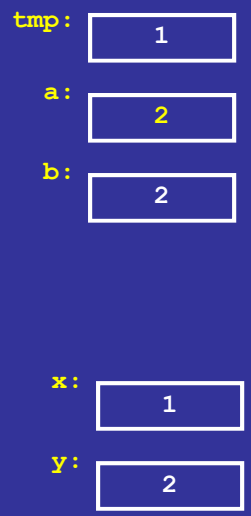
```
#include <stdio.h>

void
swap1(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
}

main()
{
    int x = 1, y = 2;

    swap1(x, y);
    printf("%d %d\n", x, y);
}
```



71

Solution 1

```
#include <stdio.h>

void
swap1(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
}

main()
{
    int x = 1, y = 2;

    swap1(x, y);
    printf("%d %d\n", x, y);
}
```



72

Solution 1

```
#include <stdio.h>

void
swap1(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
}

main()
{
    int x = 1, y = 2;

    swap1(x, y);
    printf("%d %d\n", x, y);
}
```

tmp: 1
a: 2
b: 1

x: 1
y: 2

73

Solution 1

```
#include <stdio.h>

void
swap1(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
}

main()
{
    int x = 1, y = 2;

    swap1(x, y);
    printf("%d %d\n", x, y);
}
```

tmp: 1
a: 2
b: 1

x: 1
y: 2

74

Solution 2

```
#include <stdio.h>

void
swap2(int* a, int* b)
{
   int tmp;

   tmp = *a;
   *a = *b;
   *b = tmp;
}

main()
{
   int x = 1, y = 2;

   swap2(&x, &y);
   printf("%d %d\n", x, y);
}
```

75

Solution 2

```
#include <stdio.h>

void
swap2(int* a, int* b)
{
   int tmp;

   tmp = *a;
   *a = *b;
   *b = tmp;
}

main()
{
   int x = 1, y = 2;

   swap2(&x, &y);
   printf("%d %d\n", x, y);
}
```

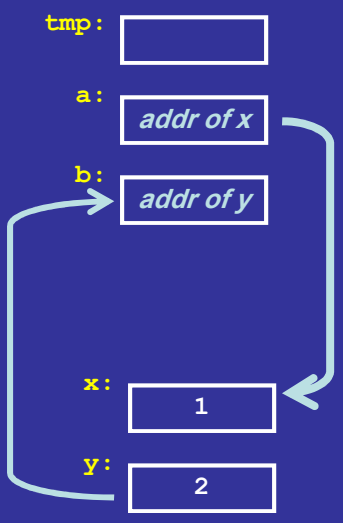
x: 1

y: 2

76

Solution 2

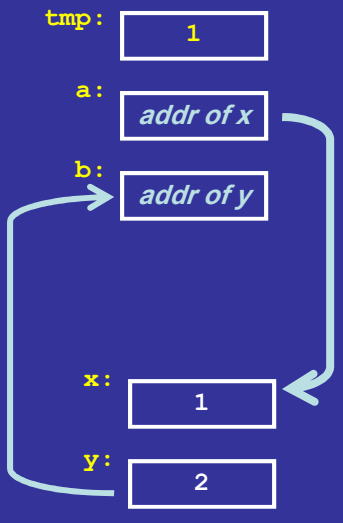
```
#include <stdio.h>

void
swap2(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
}

main()
{
    int x = 1, y = 2;

    swap2(&x, &y);
    printf("%d %d\n", x, y);
}
```



77

Solution 2

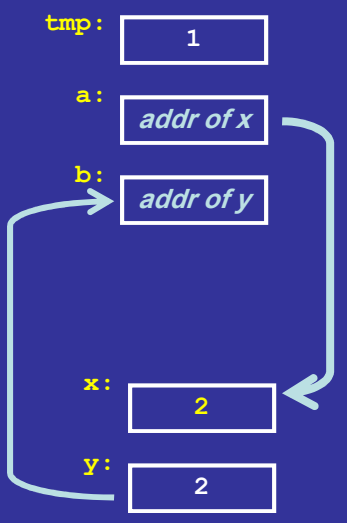
```
#include <stdio.h>

void
swap2(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
}

main()
{
    int x = 1, y = 2;

    swap2(&x, &y);
    printf("%d %d\n", x, y);
}
```



78

Solution 2

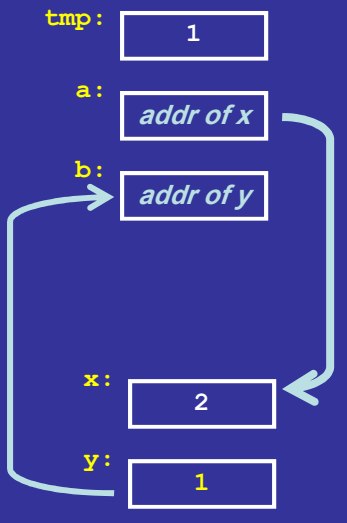
```
#include <stdio.h>

void
swap2(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
}

main()
{
    int x = 1, y = 2;

    swap2(&x, &y);
    printf("%d %d\n", x, y);
}
```



79

Solution 2

```
#include <stdio.h>

void
swap2(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
}

main()
{
    int x = 1, y = 2;

    swap2(&x, &y);
    printf("%d %d\n", x, y);
}
```



80

```
#include <stdio.h>

void
swap2(int* a, int* b)
{
   int tmp;

   tmp = *a;
   *a = *b;
   *b = tmp;
}

main()
{
   int x = 1, y = 2;

   swap2(&x, &y);
   printf("%d %d\n", x, y);
}
```

Solution 2

x: 2

y: 1

81

Consider this example:

```
void initialize (int *p, int *q)
{
      *p = 3;
      *q = 7;
}
main ()
{
   int a, b;
   initialize (&a, &b);
   /*  Arguments to initialize are
   pointers to a and b, A pointer to a is
   stored in p and pointer to b in q  */
   printf ("%d %d\n",  a, b);
   return 0;
}
```

82

Recall:

int i;

scanf("%d", &i);

- We allow **scanf** to modify variable **i** because we supply ***scanf*** with the address of **i**
- *Generally, **passing** a **Pointer** to a **Function grants the Function permission to modify the variable;***
- Develop a C code to **input width** and **height** of a **rectangle** and **calculate** the **area**, and the **perimeter**? One Solution: Three functions:
  - **read;** **// Reading the data: width & height**
  - **cal_area;** **// Calculating the area**
  - **cal_perimeter;** **// Calculating the Perimeter**

83

```
#include <stdio.h>

void read(int *x , int *y);
int calc_area(int x, int y);
int calc_perimeter(int x , int y);

int main(void)
{
  int width, height, area, perimeter;
  read(&width,&height);
  area = calc_area(width,height);
  perimeter =calc_perimeter(width,height);
  printf("\nArea is %d and perimeter is
   %d.\n", area, perimeter);
  return 0;
}
```

84

```
void read(int *x , int *y)
{
  printf("\nPlease enter width and height:");
  while(1) /* Execute forever, unless its
   body contains a     break, goto, or return
   statements */
  {
    scanf("%d %d" , x , y); // Recall: x & y
   are pointers
    if ( *x<=0 || *y<=0 || *x>60 || *y>20)
        printf("\nInvalid input Please enter
                   width & height:");
    else
        break;
  }
}
```

85

Cont:

```
int calc_area(int x, int y)
{
   return x * y;
}

int calc_perimeter(int x , int y)
{
   return ( x + y ) * 2;
}
```

86

// Calculate the cube of a variable using call-by-value

```
#include <stdio.h>

int cubeByValue( int n );

int main()
{
    int number = 5;
    printf("The original value of number is %d",
           number);
    number = cubeByValue(number); /* Pass number
                      by value to cubeByValue */
   printf("\nThe new value of number is %d\n",
          number);
   return 0;
}
   int cubeByValue( int n )
   {   return n * n * n;  }
```

The original value of number is 5
The new value of number is 125

88

```
// Cube a variable: Using call-by-reference with a pointer
argument
#include <stdio.h>
void cubeByReference( int *nPtr ); /* prototype */
main()
{
  int number = 5; /* initialize number */
  printf("The original value of number is %d",
          number);
  /* pass address of number to cubeByReference */
  cubeByReference( &number );
  printf("\nThe new value of number is %d\n",
  number);
  return 0;
}
```

Notice that the function prototype takes a pointer to an integer.

Notice how the address of `number` is given - `cubeByReference` expects a pointer (an address of a variable).

```
// Calculate cube of *nPtr; modifies variable number in main
void cubeByReference( int *nPtr )
{
    *nPtr = *nPtr  *  *nPtr  *  *nPtr;
}
```

90

The original value of number is 5
The new value of number is 125

91

Before main calls cubeByValue :

```
int main()
{
   int number = 5 ;

   number=cubeByValue(number);
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: undefined

After cubeByValue receives the call:

```
int main()
{
   int number = 5 ;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: 5

After cubeByValue cubes parameter n and before cubeByValue returns to main :

```
int main()
{
   int number = 5 ;

   number = cubeByValue( number );
}
```

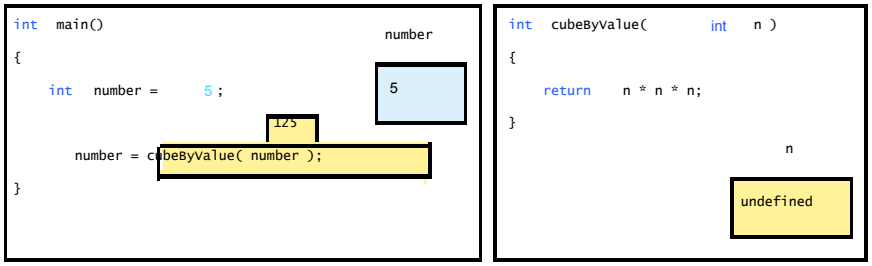
number: 5

```
int cubeByValue( int n )
{
   return 125
          n * n * n;
}
```

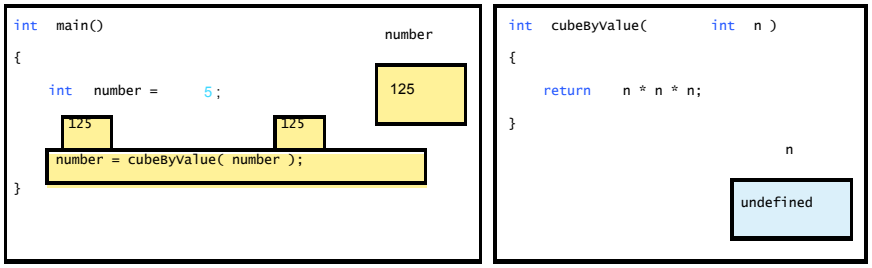
n: 5

92

After `cubeByValue` returns to `main` and before assigning the result to `number` :

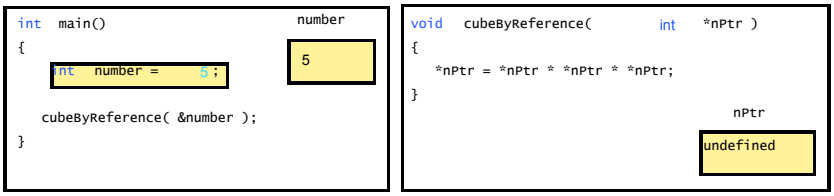


After `main` completes the assignment to `number` :

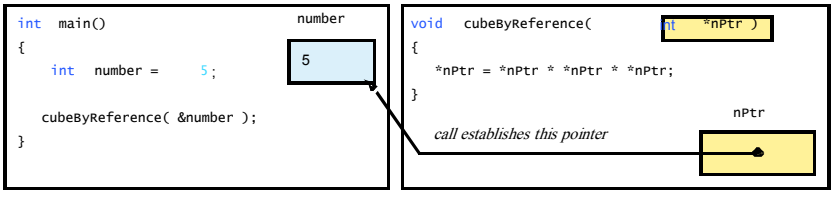


93

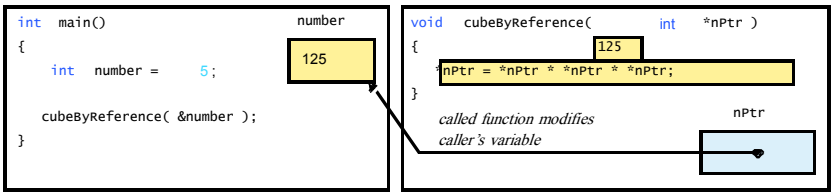
Before `main` calls `cubeByReference` :



After `cubeByReference` receives the call and before `*nPtr` is cubed:



After `*nPtr` is cubed and before program control returns to `main` :



94

- Develop a C code that **swaps** the values in two variables in memory:
  - Initialize: grade1 = 'D' and grade2 = 'A';
  - Switch the values in grade1 and grade2.

```
char grade1= 'D', grade2 = 'A', temp;
printf ("At the beginning grade1 is %c and grade2 is %c\n", grade1, grade2);
temp = grade1 ;
grade1 = grade2 ;
grade2 = temp ;
printf ("At the end grade1 is %c and grade2 is %c\n", grade1, grade2);
```

	Before Swap			After Swap		
Variable Name	grade1	grade2	temp	grade1	garde2	temp
Value	D	A	??	A	D	D
Memory Location	123456	123457	123458	123456	123457	123458

95

```
#include <stdio.h>
void SwapEm (char grade1, char grade2);
main ()
{
   char grade1= 'D', grade2 = 'A';
   printf ("Before: grade1=%c & grade2=%c\n",
            grade1,grade2);
   SwapEm (grade1, grade2);
   printf ("After grade1 =%c & grade2=%c\n", grade1,
            grade2);
}
void SwapEm (char grade1, char grade2)
{
      char temp;
      temp = grade1;
      grade1 = grade2;
      grade2 = temp;
}
```

96

```
#include <stdio.h>
void SwapEm (char *p_grade1, char *p_grade2);
main ()
{
   char grade1= 'D', grade2 = 'A';
   printf("Before: grade1=%c & grade2=%c\n", grade1,
           grade2);
   SwapEm (&grade1, &grade2);
   printf("After: grade1=%c & grade2=%c\n", grade1,
           grade2);
}
void SwapEm (char *p_grade1, char *p_grade2)
{
      char temp;
      temp = *p_grade1;
       *p_grade1 = *p_grade2;
       *p_grade2 = temp;
}
```

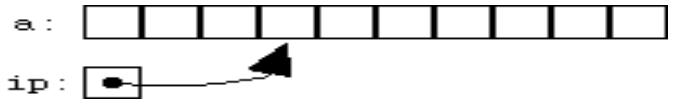
97

Review

- **A pointer:**
  - A **variable** that contains a **memory address**;
  - **Declaration: *** before the variable's name: **float *x, y, *z**;
  - **& operator:** Gets the location of a value that is loaded into a pointer variable: **x = &y;** To make x "points to" y;
  - *** operator:** When applied to a pointer it means **get the value that is pointed to by: x = &y;**
    y = 7;
    printf("%d", *x); // prints **7**
  - **Pointers** can **point** to **single variables** and **cells** of an **array**
  - 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.**

98 C_Prog. Spring 2012

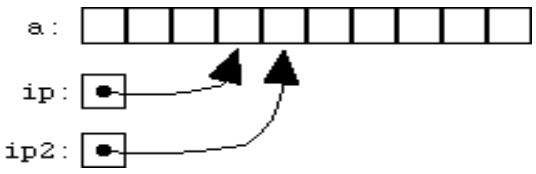
• Example:




```
int *ip;          // ip defined as pointer
int a[10];        // defines array a of 10 elements
ip = &a[3];// ip points to a[3], i.e. ip points at 4th cell of a
```

• Once we have a pointer pointing into an array, we can start doing Pointer Arithmetic

→ If *ip* is a pointer to *a[3]*, `ip = &a[3];` we can `add 1` to *ip* ➔ *ip + 1*



What does it mean to add one to a pointer?

In C, it gives a pointer to the **cell one farther on**, which in this case is *a[4]*

```
ip2 = ip + 1;
ip2 = 4;     // Sets a[4] to value of 4
```


• But it's not necessary to assign a new pointer value to a pointer variable in order to use it;

• We could compute a new pointer value & and use it immediately:

```
*(ip + 1) = 5;   // Changes a[4] again, setting it to 5
*ip+1; //Fetching & adding 1 to the value pointed to by ip;
*(ip + 1); // Accesses the value one past the one pointed
  to by ip // Paranthesis are needed: unary operator * has
  higher precedence than addition + operator;
```

If **ip** points to **a[3],** then

```
*(ip + 3) = 7;    /* sets a[6] to 7 */      and
*(ip - 2) = 4;    /* sets a[1] to 4 */
```

• We can add one to a pointer, & change the same pointer:

```
ip = ip + 1; // ip points one past where it used to
ip += 1; or ip++; //W e can increment a pointer using
```

Pointer Subtraction & Comparison

Consider the Example:

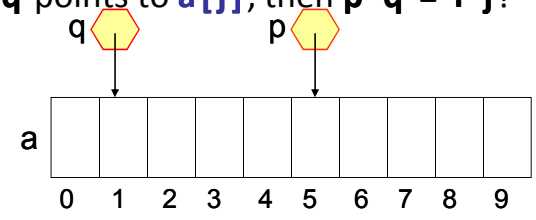
ip2 = ip1 + 3; // Which means that **ip2 - ip1 is 3**

We added an integer to a pointer to get a new pointer, pointing somewhere beyond the original *(as long as it's in the same array)*

- When you **subtract** two pointers, as long as they point into the same array, the result is the **distance** between them measured in array elements:

If **p** points to **a[i]** and **q** points to **a[j]**, then **p-q = i-j**!

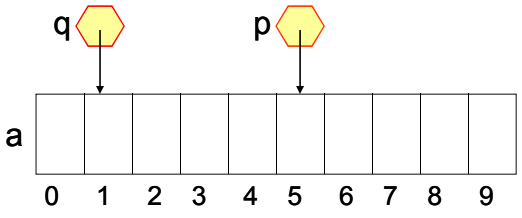
```
p = &a[5];
q = &a[1];
i = p – q; // i is 4
i = q – p; // i is –4
```



- To compare pointers, we use **relational operators (<, <=, >, >=)** & **equality operators (==, !=)**
  - Can **compare** only if pointers point to elements of the same array;
- To test for **equality** or **inequality**, the two pointers do not have to point into the same array
  - Consider:

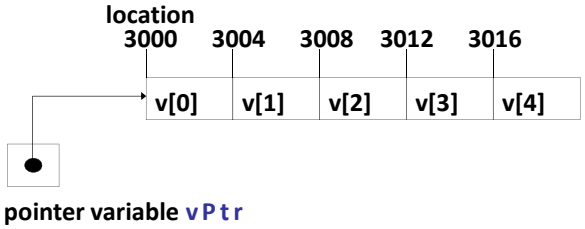
```
p = &a[5];
q = &a[1];
```

Then,




```
p <= q      /* is 0, points lower  */
p >= q      /* is 1, points higher */
```

- 5 element int array on machine with 4 byte integers.
  - **vPtr** points to first element **v[0]** at location **3000**: **(vPtr = 3000)**
  - **vPtr += 2;** /* Sets vPtr to 3008. * **vPtr** points to **v[2]** (incremented by 2), but the machine has 4 byte integers, so it points to address **3008**!*/



- Subtracting pointers Returns **number of elements** from one to the other. If: **vPtr2 = v[2];** **vPtr = v[0];** **vPtr2 - vPtr** would produce **2**

Pointers for Array Processing

- Pointer arithmetic allows you to visit the elements of an array by incrementing a pointer variable!

```
// A program segment that sums the elements of an array
#define N 10
int a[N], sum, *p;
sum = 0;
/* loop increments p, as a result pointing to a[1], a[2]
   and terminates when p goes past last element of a */
for(p = &a[0]; p < &a[N]; p++)//*p 1st points to a[0]
      sum += *p;
```

Note: **p < &a[N]** ➔ We applied address operator to **a[N]** But **a[N]** does not exist (**a** is indexed from **0** to **N-1**). Using **a[N]** in this way is **safe** Body of LOOP will be executed with **p** equal to **&a[0], &a[1], , &a[N-1]** stops at **p=&a[N]**

- Often the indirection * operator and the increment ++ operator are combined

  *p++ = 0 ; /*Sets *p to 0 before incrementing p as ++ operator takes precedence over * operator */

  (*p)++ ➔ Increments the value pointed to

  *++p ➔ Increments p before accessing the location
- Expressions of the form (ptr+1) and (ptr+2) **do not move** the pointer: they are used **to reference** elements **one** and **two steps in front**.

  ptr += 2 ➔ Is a statement that **moves** the pointer
- *(ptr++) ➔ **Returns** the current value that the pointer's pointing to, THEN the pointer is moved

  *(++ptr) ➔ The pointer is moved first.
- The same applies for decrementing


```
#include <stdio.h>
main()
{
  int *ptr;
  int arrayInts[10]={1,2,3,4,5,6,7,8,9,10};
  ptr = arrayInts; // ptr = &arrayInts[0]; is also fine
  printf(“Pointer is pointing to the first array element, ");
  printf("which is %d.\n", *ptr);
```

Pointer is pointing to the first array element, which is 1

```
  printf("Let's increment it.....\n");
```

Let's increment it.....

```
  ptr++;
  printf("Now it should point to next element,
                                        %d.\n", *ptr);
```

Now it should point to the next element, which is 2.

```
printf("But suppose we point to the 3rd  and 4th: %d %d.
                                \n", *(ptr+1),*(ptr+2));
```

But suppose we point to the 3rd and 4th: 3 4.

```
ptr += 2;
printf("Now skip the next 4 to point to the 8th: %d.\n",
  *(ptr+=4));
```

Now skip the next 4 to point to the 8th: 8.

```
ptr--;
printf("Did I miss out my lucky number %d?!\n",
  *(ptr++));
```

Did I miss out my lucky number 7?!

```
printf("Back to the 8th it is then..%d.\n", *ptr);
return 0;
```

Back to the 8th it is then..... 8.

```
}
```



```
/* Reverses a series of numbers (pointer version) */
#include <stdio.h>
#define N 10
main()
{
  int a[N], *p;
  printf("Enter %d numbers: ", N);
  for (p = a; p < a + N; p++)
    scanf("%d", p);
  printf("In reverse order:");
  for (p = a + N - 1; p >= a; p--)
    printf(" %d", *p);
  printf("\n");
  return 0;
}
```

- Arrays and pointers are closely related
  - Array name like a constant pointer
  - Pointers can do array subscripting operations
- Define an array `b[ 5 ]` and a pointer `bPtr`
  - To set them equal to one another use: `bPtr = b;`
    - Array name `b` is actually the address of the first element of array `b[ 5 ]` ➔ `bPtr = &b[ 0 ];`
    - Explicitly assigns `bPtr` to address of first element of `b`
  - Element **`b[ 3 ]`** Can be accessed by:
    - `*(bPtr+3)` ➔ **3** is the offset, **pointer/offset notation**
    - `Bptr[3]` ➔ Same as **b[3], pointer/subscript notation**
    - Performing pointer arithmetic on the array itself `*(b+3)`
  - Conclusion: **Array name is a pointer**.

- **The following example prints out the elements and addresses of a simple character array. NOTE: When printing pointers, the special conversion %p should be used.**

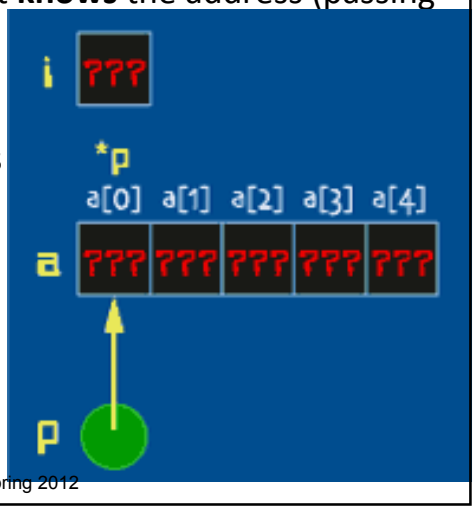
```
#include <stdio.h>
#define ARRAY_SIZE 10 /* Number of characters in array */
/* Array to print */
char array[ARRAY_SIZE] = "0123456789";
int main()
{    int index; /* Index into the array */
     for (index = 0; index < ARRAY_SIZE; ++index)
     {
        printf("&array[index]=0x%p (array+index)=0x%p
        array[index]=0x%x\n", &array[index],
        (array+index), array[index]);
      }
      return (0);
}
```

- Output:

```
&array[index]  (array+index)  array[index]
0x40b0          0x40b0          0x30
0x40b1          0x40b1          0x31
0x40b2          0x40b2          0x32
0x40b3          0x40b3          0x33
0x40b4          0x40b4          0x34
0x40b5          0x40b5          0x35
0x40b6          0x40b6          0x36
0x40b7          0x40b7          0x37
0x40b8          0x40b8          0x38
0x40b9          0x40b9          0x39
```


Arrays, Pointers, and Functions!

- Array name is a pointer
- A pointer can be passed to a function
- Function can access data as it **knows** the address (passing by reference)
- Array name is no difference!
- We can pass an array such as **a** or **b** to a function by mentioning the array name in the function call.



- **The following example determins the number of elements in an array before the value zero using poiters**

```
#include <stdio.h>
int array[] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
int *array_ptr;
int main()
{
   array_ptr = array;
   while ((*array_ptr) != 0)
      ++array_ptr;
   printf("Number of elements before zero %d\n",
          array_ptr - array);
   return (0);
}
```


Passing Arrays to Functions

- Imagine a function **dump** that accepts an array of integers as a parameter and prints the contents of the array

  `void dump(int a[],int loa)`→loa (lengthofarray): usually a global constant, loa in 2nd param. is not necessary

  ```
  { int i; for (i=0; i < loa; i++)
    printf("%d\n",a[i]); }
  ```

  OR:

  ```
  void dump(int *p, int loa)/* Pointer is passed rather thatn the Contents of the array*/
  { int i; for (i=0; i < loa; i++)
    printf("%d\n",*p++);  }
  ```

- Passing the array to the function can be done using a statement similar to:

  ```
  dump(a, 25);
  ```

- **a** is the array name, **25** is its length.

```
/* Using subscripting and pointer notations with arrays */
#include <stdio.h>
main()
{
  int b[]={10,20,30,40}; //initialize array b
  int *bPtr = b; //set bPtr to point to array b
   int i, offset;       //Two counters
  // output array b using array subscript notation
  printf( "Array b printed with:\n Array subscript notation\n" );
  for ( i = 0; i < 4; i++ ) {
    printf( "b[ %d ] = %d\n", i, b[ i ] ); }
/* output array b using array name and pointer/offset notation */
  printf( "\nPointer/offset notation where\n
        "the pointer is the array name\n" );
  for ( offset = 0; offset < 4;
                          offset++ ){
    printf("*( b + %d ) = %d\n",
            offset, *(b + offset));}
```

Array b printed with:
Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40


```
/* output array b using bPtr and array subscript notation */
  printf( "\nPointer subscript notation\n" );
  for ( i = 0; i < 4; i++ ) {
     printf( "bPtr[ %d ] = %d\n", i, bPtr[ i ] );
    }


 /* output array b using bPtr and pointer/offset notation */
  printf( "\nPointer/offset notation\n" );
  for ( offset = 0; offset < 4; offset++ ) {
     printf( "*( bPtr + %d ) = %d\n", offset, *( bPtr + offset ) );
    }
  return 0;
 }
```

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40

```
// Copying a string using array notation and pointer notation.
 #include <stdio.h>
 void copy1( char *s1, const char *s2 ); /* prototype */
 void copy2( char *s1, const char *s2 ); /* prototype */
 main()
 {
   char string1[ 10 ];       /* create array string1 */
  char *string2 = "Hello";    /* create a pointer to a string */
  char string3[ 10 ];       /* create array string3 */
  char string4[] = "Good Bye"; // create a pointer to a string
  copy1( string1, string2 );
    printf( "string1 = %s\n", string1 );
    copy2( string3, string4 );
    printf( "string3 = %s\n", string3 );
    return 0;       }
```

```
string1 = Hello
string3 = Good Bye
```



```
 /* copy s2 to s1 using array notation */
 void copy1( char *s1, const char *s2 )
   {
    int i; /* counter */
    for (i=0; (s1[i]=s2[i]) !='\0'; i++) {//loop through strings
         ; /* do nothing in body */
    } /* end for */
  } /* end function copy1 */
/* copy s2 to s1 using pointer notation */
 void copy2( char *s1, const char *s2 )
   {
    for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ ) {
      ; /* do nothing in body */
   } /* end for */
 } /* end function copy2 */
```