CS 2211
Systems Programming

Part Six:

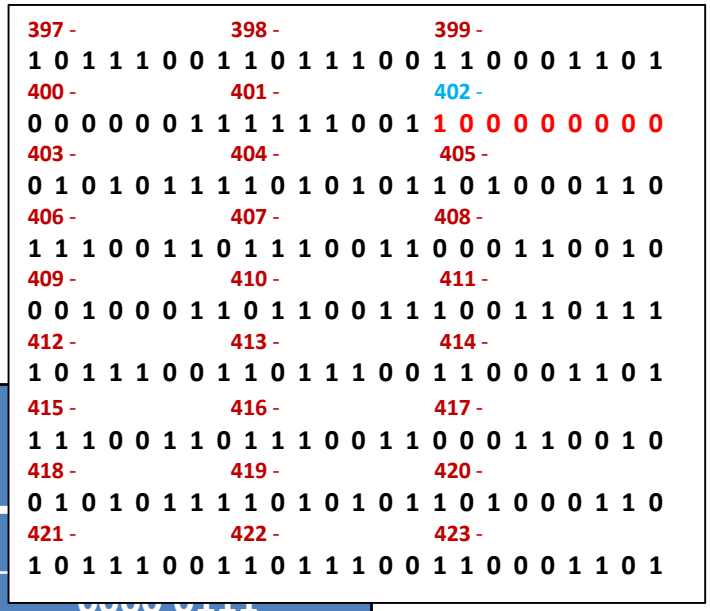
Pointers

POINTERS

… nothing more than a mechanism to manipulate and utilize computer memory

… VERY EASY:

variable + **address** == **pointer**



Label	Address	Value	
	399		
a	400	7	[illegible]
b	401	-13	1111 0011
c	402	0	0000 0000
	403		
	404		
	405		
	406		
	…		

POINTERS

… nothing more than a mechanism to
manipulate and utilize computer memory

- nothing more than just another variable
  - BUT instead of a value – it stores an **address** to another variable

* (asterisk) – declare a pointer variable of a type
* (asterisk) – what the address points to [indirection]
(go to the value 'box' of the address stored in this value)

& (ampersand) – return the address of the variable.

POINTERS

… nothing more than a mechanism to
manipulate and utilize computer memory

- **Declaration**
  variable type (asterisk) pointer variable name
  - all pointer variables are same size (same number of bits)

    conventional memory (DOS) (2.5 bytes – 20 bits)
    (what is the limit in size of available memory?)
    2^{20} = 1,048,576 (~ 1 MB) [640 kb of usable memory]

    based on OS and architecture (assume 4 bytes – 32 bits)
    (what is the limit in size of available memory?)
    2^{32} = 4,294,967,295 (~ 4 Gigs)

    based on OS and architecture (assume 8 bytes – 64 bits)
    (what is the limit in size of available memory?)
    2^{64} = 18,446,744,073,709,551,616 (~ 16 EiB [exabytes])

POINTERS

… nothing more than a mechanism to
manipulate and utilize computer memory

- for the remainder of this class: Assume 32 bits

When a pointer variable is declared
- the variable name must be preceded by an asterisk:

```
int *p;
```

`p` in this incarnation is a pointer variable capable of pointing to **objects** of type **int**.

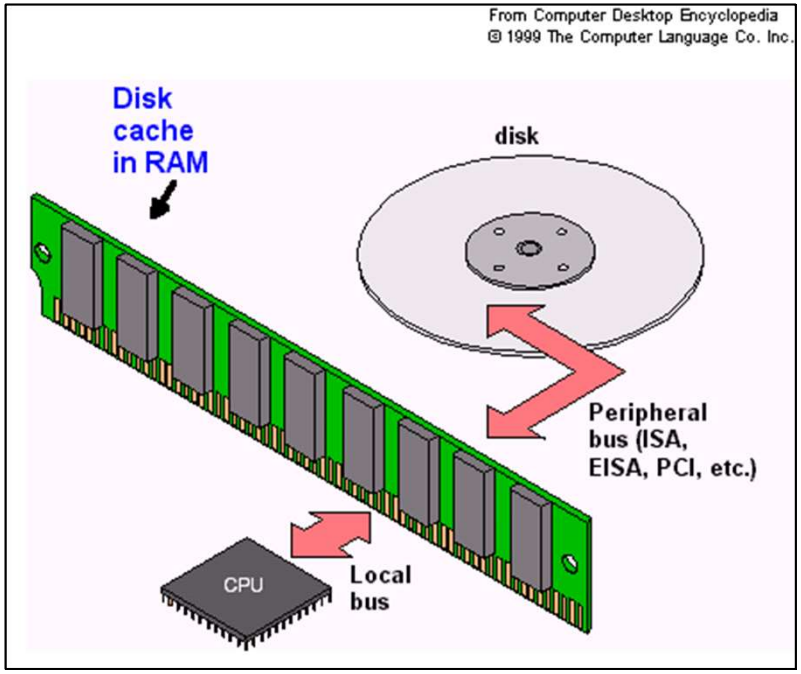
- why not a “pointer” variable type instead ?
- **pointer arithmetic** –the computer must know what the data type is

Computer Definition

Proccessing

RAM (Random Access Memory)

- ‘Waiting Room’ of instructions



POINTERS

… nothing more than a mechanism to
manipulate and utilize computer memory

- **Declaration**

Pointer variables can appear in declarations along with other variables:

```
int i, j, a[10], b[20], *p, *q;
```

C requires that every pointer variable point only to:
objects of a particular type (the **referenced** type):

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

There are no restrictions on what the referenced type may be.

POINTERS

… nothing more than a mechanism to
manipulate and utilize computer memory

* (asterisk) – indirection:
go to the **memory location** stored in this variable and return the value from that location.

& (ampersand) – return the **address** of the variable.

Pointers in C

END OF PART 1

POINTERS

```
char c;    /* 1 byte */
```

Type	Label	Address	Value	Binary
		399		
char	c	400		
		401		
		402		
		403		
		404		
		405		
		406		
		…		

POINTERS

```
char c;    /* 1 byte */
c = 37;    /* assignment */
           /* at address of c (400)
              switch the bits to 00100101
              i.e. 37 in base 10 */
```

Type	Label	Address	Value	Binary
		399		
char	c	400	37	0010 0101
		401		
		402		
		403		
		404		
		405		
		406		
		…		

POINTERS

```
char c;    /* 1 byte */
char *cp   /* a pointer of type char 4 bytes */
```

Type	Label	Address	Value	Binary
		399		
char	c	400		
p to char	cp	401		
		402		
		403		
		404		
		405		
		406		
		…		

POINTERS

```
char c;    /* 1 byte */
char *cp   /* a pointer of type char 4 bytes */
```

Type	Label	Address	Value
		399	
char	c	400	
p to char	cp	401 - 404	
		...	

POINTERS

```
char c;    /* 1 byte */
char *cp   /* a pointer of type char 4 bytes */
```

WARNING:
Declaring a pointer variable sets aside space for a pointer
but doesn't make it point to an object:

```
char *cp;  /* points nowhere */
```

It's crucial to define **cp** before we use it.

Type	Label	Address	Value
		399	
char	c	400	
p to char	cp	401 - 404	
		...	

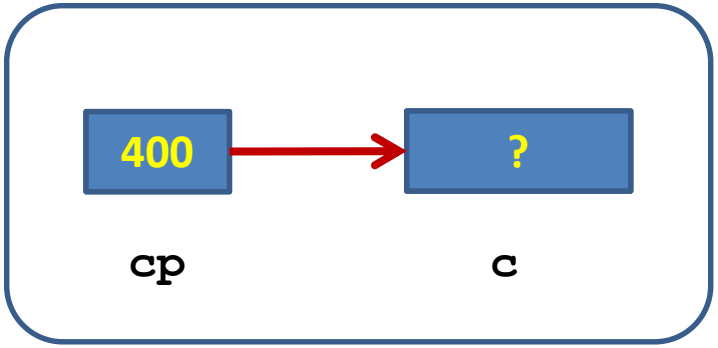
POINTERS

```
char c;    /* 1 byte */
char *cp   /* a pointer of type char 4 bytes */

cp = &c;   /* cp is assigned the ADDRESS of c */
```

Type	Label	Address	Value
		399	
char	c	400	
p to char	cp	401 - 404	400
		...	

POINTERS

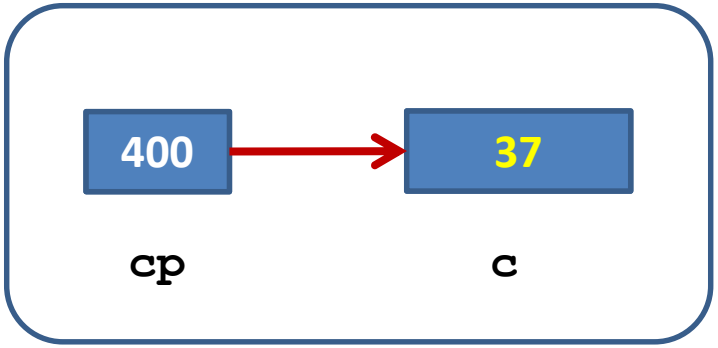
```
char c;    /* 1 byte */
char *cp   /* a pointer of type char 4 bytes */

cp = &c;   /* cp is assigned the ADDRESS of c */
```



Type	Label	Address	Value
char	c	400 -	
p to char	cp	401 - 404	400

POINTERS

```
char c;    /* 1 byte */
char *cp   /* a pointer of type char 4 bytes */

cp = &c;   /* cp is assigned the ADDRESS of c */
*cp = 37; /* in the location stored in cp */
```



Type	Label	Address	Value
char	c	400 -	37
p to char	cp	401 - 404	400

POINTERS

```
char c;    /* 1 byte */
char *cp   /* a pointer of type char 4 bytes */

cp = &c;   /* cp is assigned the ADDRESS of c */
*cp = 37; /* in the location stored in cp */
printf("c = %d and *cp = %d and %u\n", c, *cp, cp);
```

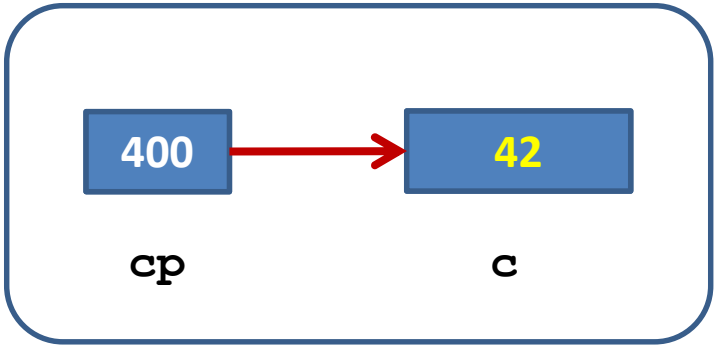
本体 value of c. Address of c.

Type	Label	Address	Value
char	c	400 -	37
p to char	cp	401 - 404	400

output:
37 and 37 and 400

POINTERS

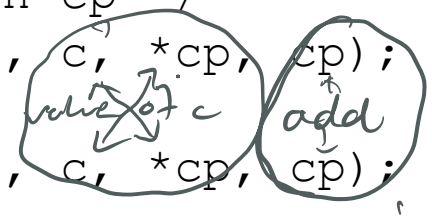
```
char c;    /* 1 byte */
char *cp   /* a pointer of type char 4 bytes */

cp = &c;   /* cp is assigned the ADDRESS of c */
*cp = 37; /* in the location stored in cp */
printf("c = %d and *cp = %d and %u\n", c, *cp, cp);
c = 42;
```



Type	Label	Address	Value
char	c	400 -	42
p to char	cp	401 - 404	400

POINTERS

```
char c;    /* 1 byte */
char *cp   /* a pointer of type char 4 bytes */

cp = &c;   /* cp is assigned the ADDRESS of c */
*cp = 37; /* in the location stored in cp */
printf("c = %d and *cp = %d and %u\n", c, *cp, cp);
c = 42;
printf("c = %d and *cp = %d and %u\n", c, *cp, cp);
```



output:
37 and 37 and 400
42 and 42 and 400

Type	Label	Address	Value
char	c	400 -	42
p to char	cp	401 - 404	400

Pointers in C

END OF PART 2

POINTERS

```
char c,*cp;        /* 1 byte  – 4 bytes */
```

Type	Label	Address	Value
char	c	400 -	
p to char	cp	401 - 404	

POINTERS

```
char c,*cp;        /* 1 byte  – 4 bytes */
int i,*ip;         /* 4 bytes – 4 bytes */
```

Type	Label	Address	Value
char	c	400 -	
p to char	cp	401 - 404	
int	i	405 - 408	
p to int	ip	409 - 412	

POINTERS

```
char c,*cp;        /* 1 byte  – 4 bytes */
int i,*ip;         /* 4 bytes – 4 bytes */
float f,*fp;       /* 4 bytes – 4 bytes */
```

Type	Label	Address	Value
char	c	400 -	
p to char	cp	401 - 404	
int	i	405 - 408	
p to int	ip	409 - 412	
float	f	413 – 416	
p to float	fp	417 – 420	

POINTERS

length of pointers are 4 bytes.

```
char c,*cp;       /* 1 byte  - 4 bytes */
int i,*ip;        /* 4 bytes - 4 bytes */
float f,*fp;      /* 4 bytes - 4 bytes */
double d,*dp;     /* 8 bytes - 4 bytes */
```

Type	Label	Address	Value
char	c	400 -	
p to char	cp	401 - 404	
int	i	405 - 408	
p to int	ip	409 - 412	
float	f	413 – 416	
p to float	fp	417 – 420	
double	d	421 – 428	
p to doubl	dp	429 - 432	

POINTERS (assume a 64 bit architecture)

```
double a;
double *ap;
float b;
float *bp;
int c;
int *cp;
char d;
char *dp;
```

Type	Label	Address	Value
double	a	400 - ?	
p to doubl	ap	???	
float	b	???	
p to float	bp	???	
int	c	???	
p to int	cp	???	
char	d	???	
p to char	dp	???	

POINTERS

```
char c,*cp;        /* 1 byte  - 4 bytes */
int i,*ip;         /* 4 bytes - 4 bytes */
float f,*fp;       /* 4 bytes - 4 bytes */
double d,*dp;      /* 8 bytes - 4 bytes */

cp = &c;
```

cp → c .

Type	Label	Address	Value
char	c	400 -	
p to char	cp	401 - 404	400
int	i	405 - 408	
p to int	ip	409 - 412	
float	f	413 – 416	
p to float	fp	417 – 420	
double	d	421 – 428	
p to doubl	dp	429 - 432	

POINTERS

```
char c,*cp;        /* 1 byte  - 4 bytes */
int i,*ip;         /* 4 bytes - 4 bytes */
float f,*fp;       /* 4 bytes - 4 bytes */
double d,*dp;      /* 8 bytes - 4 bytes */

cp = &c;
ip = &i;
```

ip -> i.

Type	Label	Address	Value
char	c	400 -	
p to char	cp	401 - 404	400
int	i	405 - 408	
p to int	ip	409 - 412	405
float	f	413 – 416	
p to float	fp	417 – 420	
double	d	421 – 428	
p to doubl	dp	429 - 432	

POINTERS

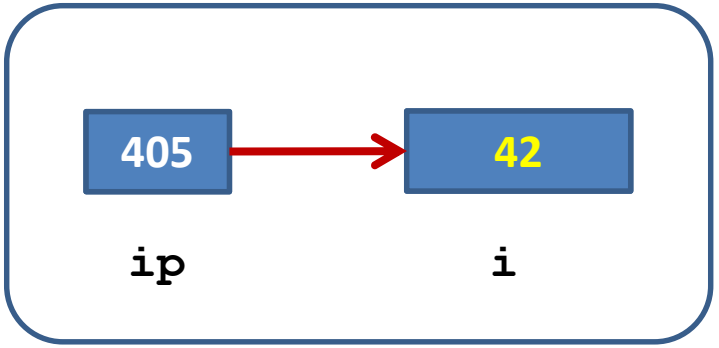
```
char c,*cp;       /* 1 byte  - 4 bytes */
int i,*ip;        /* 4 bytes - 4 bytes */
float f,*fp;      /* 4 bytes - 4 bytes */
double d,*dp;     /* 8 bytes - 4 bytes */

cp = &c;
ip = &i;
*ip = 42;
```

*ip = i = 42.

Type	Label	Address	Value
char	c	400 -	
p to char	cp	401 - 404	400
int	i	405 - 408	42
p to int	ip	409 - 412	405
float	f	413 – 416	
p to float	fp	417 – 420	
double	d	421 – 428	
p to doubl	dp	429 - 432	

POINTERS

```
char c,*cp;        /* 1 byte  - 4 bytes */
int i,*ip;         /* 4 bytes - 4 bytes */
float f,*fp;       /* 4 bytes - 4 bytes */
double d,*dp;      /* 8 bytes - 4 bytes */

cp = &c;
ip = &i;
*ip = 42;
```



Type	Label	Address	Value
char	c	400 -	
p to char	cp	401 - 404	400
int	i	405 - 408	42
p to int	ip	409 - 412	405
float	f	413 – 416	
p to float	fp	417 – 420	
double	d	421 – 428	
p to doubl	dp	429 - 432	

POINTERS

```
char c,*cp;        /* 1 byte  - 4 bytes */
int i,*ip;         /* 4 bytes - 4 bytes */
float f,*fp;       /* 4 bytes - 4 bytes */
double d,*dp;      /* 8 bytes - 4 bytes */

cp = &c;
ip = &i;
*ip = 42;
```

NOTE:
As long as **`ip`** points to **`i`**, **`*ip`** is an ***alias*** for **`i`**.

`*ip` has the same value as **`i`.**

Changing the value of **`*ip`** changes the value of **`i`**.

pe	Label	Address	Value
char	c	400 -	
to char	cp	401 - 404	400
int	i	405 - 408	42
to int	ip	409 - 412	405
float	f	413 – 416	
to float	fp	417 – 420	
ouble	d	421 – 428	
p to doubl	dp	429 - 432	

POINTERS

```
char c,*cp;        /* 1 byte  - 4 bytes */
int i,*ip;         /* 4 bytes - 4 bytes */
float f,*fp;       /* 4 bytes - 4 bytes */
double d,*dp;      /* 8 bytes - 4 bytes */

cp = &c;
ip = &i;
*ip = 42;
int j = *&i;
```

NOTE:
Applying **&** to a variable produces a pointer to the variable.
Applying ***** to the pointer takes us back to the original variable:

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

Type	Label	Address	Value
char	c	400 -	
to char	cp	401 - 404	400
int	i	405 - 408	42
to int	ip	409 - 412	405
float	f	413 – 416	
to float	fp	417 – 420	
double	d	421 – 428	
to doubl	dp	429 - 432	
int	j	433 - 436	42

POINTERS

```
char c,*cp;        /* 1 byte  - 4 bytes */
int i,*ip;         /* 4 bytes - 4 bytes */
float f,*fp;       /* 4 bytes - 4 bytes */
double d,*dp;      /* 8 bytes - 4 bytes */

cp = &c;
printf("%d", *ip); /* WRONG */
printf("%d", &ip); /* CORRECT */
```

WARNING:

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

```
int *ip;
printf("%d", *ip); /* WRONG */
printf("%d", &ip); /*CORRECT*/
```

Type	Label	Address	Value
char	c	400 -	
p to char	cp	401 - 404	400
int	i	405 - 408	
p to int	ip	409 - 412	
float	f	413 – 416	
p to float	fp	417 – 420	
double	d	421 – 428	
p to doubl	dp	429 - 432	

POINTERS

```
char c,*cp;        /* 1 byte  - 4 bytes */
int i,*ip;         /* 4 bytes - 4 bytes */
float f,*fp;       /* 4 bytes - 4 bytes */
double d,*dp;      /* 8 bytes - 4 bytes */

cp = &c;
ip = 137;
```

WARNING:

never assign a value to a pointer variable!

```
ip = 137;
/* assumes 137 is an address */
```

pointer variables must only contain addresses.

Type	Label	Address	Value
char	c	400 -	
p to char	cp	401 - 404	400
int	i	405 - 408	
p to int	ip	409 - 412	137
float	f	413 – 416	
p to float	fp	417 – 420	
double	d	421 – 428	
p to doubl	dp	429 - 432	

POINTERS

```
char c;    /* 1 byte */
char *cp   /* a pointer of type char 4 bytes */

c = 7;
cp = &c;
printf("%d  %u  %p  value = %d\n", cp, cp, cp, *cp);
/*
    %d -> int
    %u -> unsigned int
    %p -> pointer address
          (in hex)
*/
```

Label	Address	Value
c	3221202422	7
cp	3221202423 - 3221202426	3221202422

output:
-1073764872 3221202422 0xbfffa5f6 value = 7

Note: address of c is: 1011 1111 1111 1111 1010 0101 1111 0110

Pointers in C

END OF PART 3

POINTERS

```
char ca[3],*cap;        /* 3 x 1 byte  – 4 bytes */
```

- a pointer variable can hold the **address of any variable**, including a cell in an array

- a pointer variable is **NEVER** used to hold anything BUT an **address**.

- a pointer variable is **NEVER** used to hold **actual data**.

Label	Address	Value
ca[0]	400 -	
ca[1]	401 -	
ca[2]	402 -	
cap	403 - 406	

POINTERS

```
char ca[3],*cap;        /* 3 x 1 byte  - 4 bytes */
cap = &(ca[1]);
```

note:

`&(ca[1])` is preferred over `&ca[1]`

(order of precedent is shown using this)

Label	Address	Value
ca[0]	400 -	
ca[1]	401 -	
ca[2]	402 -	
cap	403 - 406	401

POINTERS

```
char ca[3],*cap;        /* 3 x 1 byte  - 4 bytes */
cap = &(ca[1]);
*cap = 7;
```

note:

`&(ca[1])` is preferred over `&ca[1]`

(order of precedent is shown using this)

Label	Address	Value
ca[0]	400 -	
ca[1]	401 -	7
ca[2]	402 -	
cap	403 - 406	401

POINTERS

```
char ca[3];      /* 3 x 1 bytes */
char *cp;        /* 4 bytes */
int ia[3];       /* 3 x 4 bytes */
int *ip;         /* 4 bytes */
```

Pointer Arithmetic

- address plus (+) an integer n

- advance to the nth
  memory location
  (based on the variable type)

Array	Label	Address	Value
ca	ca[0]	400 -	
	ca[1]	401 -	
	ca[2]	402 -	
	cp	403 - 406	
ia	ia[0]	407 - 410	
	ia[1]	411 - 414	
	ia[2]	415 - 418	
	ip	419 - 422	

POINTERS

```
char ca[3];      /* 3 x 1 bytes */
char *cp;        /* 4 bytes */
int ia[3];       /* 3 x 4 bytes */
int *ip;         /* 4 bytes */

cp = &(ca[0]);
ip = &(ia[0]);
```

Pointer Arithmetic

- address plus (+) an integer n

- advance to the nth
  memory location
  (based on the variable type)

Array	Label	Address	Value
ca	ca[0]	400 -	
	ca[1]	401 -	
	ca[2]	402 -	
	cp	403 - 406	400
ia	ia[0]	407 - 410	
	ia[1]	411 - 414	
	ia[2]	415 - 418	
	ip	419 - 422	407

POINTERS

```
char ca[3]        /* 3 x 1 bytes */
char *cp;         /* 4 bytes */
int ia[3],        /* 3 x 4 bytes */
int *ip;          /* 4 bytes */

cp = &(ca[0]);
ip = &(ia[0]);

*(cp+2) = 8;      // cp + 2 of what?
*(ip+2) = 33;     // ip + 2 of what?
```

Pointer Arithmetic

- plus two UNITS of the variable type

cp+2 // plus two characters (2 x 1 bytes)

ip+2 // plus two integers (2 x 4 bytes)

```
*(cp+2) = 8;      // 400 + 2 = 402
*(ip+2) = 33;     // 407 + 8 = 415
```

Array	Label	Address	Value
ca	ca[0]	400 -	
	ca[1]	401 -	
	ca[2]	402 -	8
	cp	403 - 406	400
ia	ia[0]	407 - 410	
	ia[1]	411 - 414	
	ia[2]	415 - 418	33
	ip	419 - 422	407

POINTERS

```
int iarr[3];
int *bp;
char carr[3];
char *ap;

bp = &(iarr[1]);
ap = &(carr[0]);

*(ap+2) = 8;
*(bp+1) = 33;
```

Array	Label	Address	Value
iarr	iarr[0]	400 - 403	
	iarr[1]	404 - 407	
	iarr[2]	408 - 411	33
	ap	412 - 419	420
carr	carr[0]	420	
	carr[1]	421	
	carr[2]	422	8
	bp	423 - 430	404

Pointer Arithmetic

- plus two UNITS of the variable type

ap+2 // plus two characters (2 x 1 bytes)

bp+1 // plus one integers (1 x 4 bytes)

```
*(ap+2) = 8;     // 420 + 2 = 422
*(bp+1) = 33;    // 404 + 4 = 408
```

POINTERS (assume a 64 bit architecture)

```
char *bp;
char carr[3];
int *ap;
int iarr[3];

ap = &(iarr[1]);
bp = &(carr[0]);

*(ap+1) = 8;
*(bp+2) = 33;
```

Array	Label	Address	Value
	bp	400 - ?	??KK??
	?AA?	?	??MM??
	?BB?	?	??NN??
	?CC?	?	??PP??
	?DD?	?	??SS??
	?EE?	?	??TT??
	?FF?	?	??UU??
	?GG?	?	??XX??

Pointers in C

END OF PART 4

POINTERS - special case: VOID POINTER

```
char c              /* 1 byte */
char *cp;           /* 4 bytes */
double db,          /* 8 bytes */
double *pdb;        /* 4 bytes */
void *pV ;          /* 4 bytes */

c = 'k';
cp = &c;
pdb = &db;
```

Label	Address	Value
c	400 -	'k'
cp	403 - 406	400
db	407 - 414	
pdb	415 - 418	407
pV	419 - 422	

VOID POINTER

A void pointer in c is called a **generic pointer**, it has no associated data type.

It can store the address of **any type** of object and it can be **type-casted** to any types.

POINTERS - special case: VOID POINTER

```
char c              /* 1 byte */
char *cp;           /* 4 bytes */
double db,          /* 8 bytes */
double *pdb;        /* 4 bytes */
void *pV ;          /* 4 bytes */

c = 'k';
cp = &c;
pdb = &db;

//Assigning address of character
pV = &c;

//dereferencing void pointer with character typecasting
printf("c = %c\n\n",*((char*)pV));
```

Label	Address	Value
c	400 -	'k'
cp	403 - 406	400
db	407 - 414	
pdb	415 - 418	407
pV	419 - 422	400

POINTERS - special case: VOID POINTER

```
char c              /* 1 byte */
char *cp;           /* 4 bytes */
double db,          /* 8 bytes */
double *pdb;        /* 4 bytes */
void *pV ;          /* 4 bytes */

db = 9.62312f;
cp = &c;
pdb = &db;
```

Label	Address	Value
c	400 -	
cp	403 - 406	400
db	407 - 414	9.62312
pdb	415 - 418	407
pV	419 - 422	407

```
//Assigning address of character
pV = &c;

//dereferencing void pointer with character typecasting
printf("c = %c\n\n",*((char*)pV));

//Assigning address of double
pV = &db;

//dereferencing void pointer with integer typecasting
printf("db = %lf\n\n",*((double *)pV));
```

Pointers in C

END OF PART 5

POINTERS

Referencing (&) in scanf()

```
int main( int argc, char* argv[] )
  {
     int a;
     float y;

     printf( "Enter an integer value:");
     scanf( "%d", &a );
     printf( "a = %d\n", a);

     printf( "Enter a floating point value:");
     scanf( "%f", &y );
     printf( "y = %f\n", y);

     return (0);
  }
```

POINTERS

Referencing (&) in scanf()

```
int main( int argc, char* argv[] )
  {
     int a;
     float y;

     printf( "Enter an integer value
     scanf( "%d", &a );
     printf( "a = %d\n", a);

     printf( "Enter a floating point
     scanf( "%f", &y );
     printf( "y = %f\n", y);

     return (0);
  }
```

Label	Address	Value	Binary
	399		
a	400	37	0000 0000
	401		0000 0000
	402		0000 0000
	403		001 0001
y	404	3.14159	0000 0100
	405		1100 1011
	406		0010 1111
	407		000 00000
	408		
	409		
	410		
	411		
	412		
	413		
	414		
	415		
	416		
	417		
	418		
	…		

detail:

```
scanf( "%d", &a );
```

read in a decimal value **(%d)** and place that value in the memory location **(&)** delineated by the variable labeled '**a**'.

POINTERS

… nothing more than a mechanism to
manipulate and utilize computer memory

POINTERS ARE NOT A VERY DIFFICULT TOOL TO MASTER

- but a big headache for beginning programmers

- so: why use them

one big reason: passing values to/from functions

```
int division(int numerator, int denominator,
             int *dividend, int *remainder)
```

Passing Values TO and FROM a Function

```
#include <stdio.h>

int division(int numerator, int denominator,
             int *dividend, int *remainder)
{
    printf("address stored in dividend: %u\n",dividend);
    printf("address stored in remainder: %u\n",remainder);
    if (denominator < 1)
        return(0);
    *dividend=numerator/denominator;
    *remainder=numerator%denominator;
}


int main(int argc, char *argv[])
{
     int x,y,d,r;

     x=9;
     y=2;
     printf("address of d: %u\n",&d);
     printf("address of r: %u\n",&r);
     division(x,y,&d,&r);
     printf("%d/%d = %d with %d remainder\n",x,y,d,r);
     printf("x=%d\n",x);
}
```

```
#include <stdio.h>

int division(int numerator, int denominator,
             int *dividend, int *remainder)
{
    printf("address stored in dividend: %u\n",dividend);
    printf("address stored in remainder: %u\n"          der);
    if (denominator < 1)
        return(0);
    *dividend=numerator/denomina
    *remainder=numerator%d
}


int main(int arg                gv[])
{
     int x,y,d,r;

     x=9;
     y=2;
     printf("address of d: %u\n",&d);
     printf("address of r: %u\n",&r);
     division(x,y,&d,&r);
     printf("%d/%d = %d with %d remainder\n",x,y,d,r);
     printf("x=%d\n",x);
}
```

but we will return to this later ….
for now, back to basic C

Pointers in C

END OF PART 6