Short Course on Programming in C/C++

Organized by Onur Pekcan

Contributor Selim Temizer   Instructor Hasan Yılmaz

MITENG
MITAS ENGINEERING INC.
Eski Güvercinlik Yolu No : 113 Gazi 06560 Ankara TURKIYE
P: + 90 312 296 2000 :: f: + 90 312 296 2999
www.miteng.com

Week 1 – Lecture3

Today

We will cover;

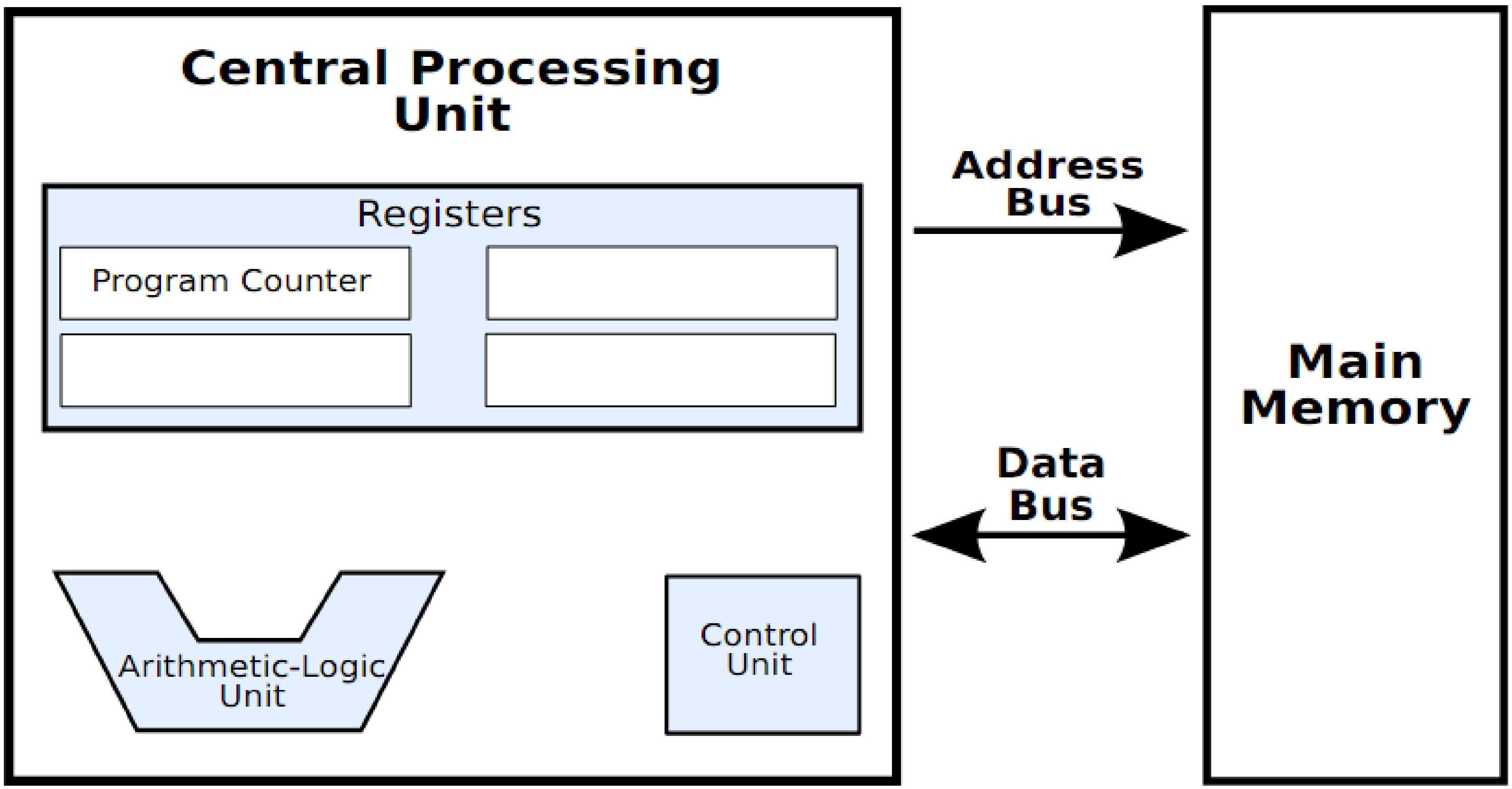
- **Arrays and Pointers**
  - Basic of Pointers
  - Array-Pointer Referencing Duality
  - Strings
  - Dynamic Memory Management
  - Function and Pointers(call-by-reference)
  - Multidimensional Arrays and Pointers
  - Pointers to Pointers
  - Pointer to Functions

Arrays and Pointers

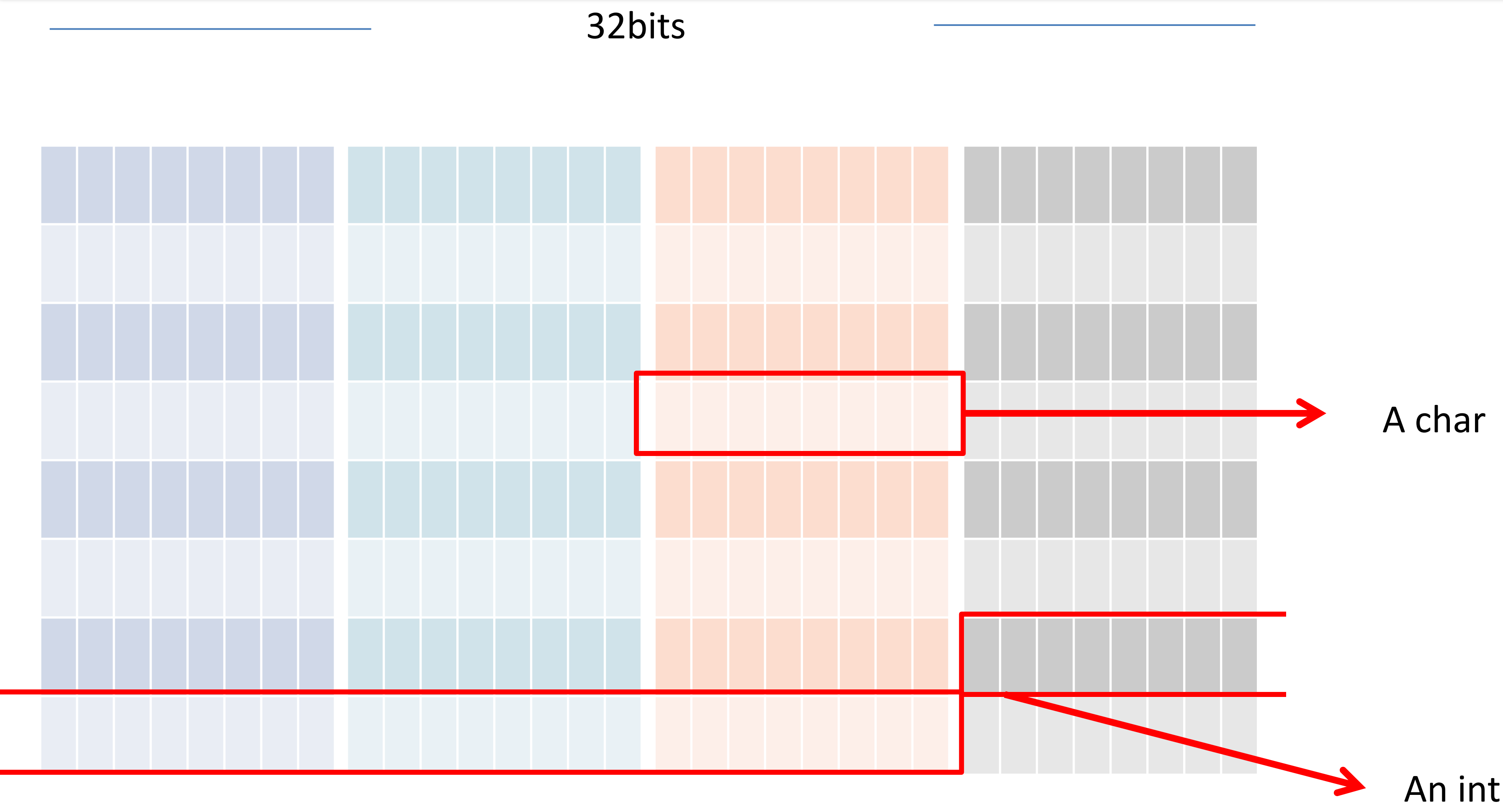
Why do we need pointers?

- For dynamic memory management!
    1. If you don't know the amount of data that your program will process, you need pointers!
    2. If your program requires a lot of deletions/insertions of new data, you might want to use linked lists and hence, you need pointers!
- Comparing data / objects / functions:
    1. You can check whether two entities are the same by comparing their addresses, for example (note that if the size of the data are different, this might not work)
- Better control over the memory!

A Brief Summary of the Von Neumann Architecture



A Brief Summary of the Von Neumann Architecture



Memory is byte addressable!

A Brief Summary of the Von Neumann Architecture

- Then, it is more convenient to view memory as a single array of 8-bit data.

Basics of Pointers

- Pointer definitions
  - `*` used with pointer variables
    - `int *myPtr;`
  - Defines a pointer to an `int` (pointer of type `int *`)
  - Multiple pointers require using a `*` before each variable definition
    - `int *myPtr1, *myPtr2;`
  - Can define pointers to any data type
  - Initialize pointers to `0`, `NULL`, or an address
    - `0` or `NULL` – points to nothing (`NULL` preferred)

Good Programming Practice

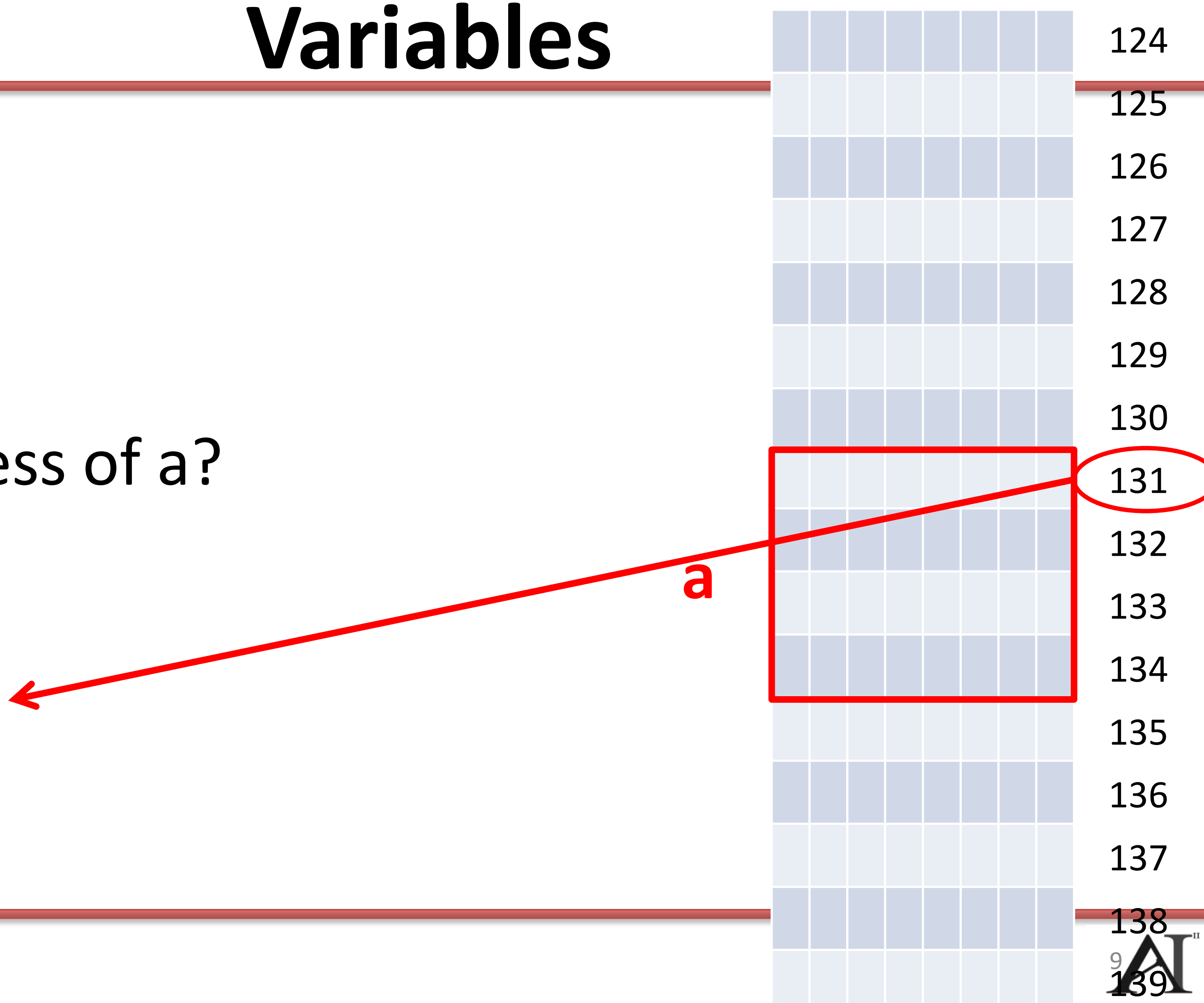
- Include the letters `ptr` in pointer variable names to make it clear that these variables are pointers and thus need to be handled appropriately

- Initialize pointers to prevent unexpected results.

Memory & Data & Addresses & Variables

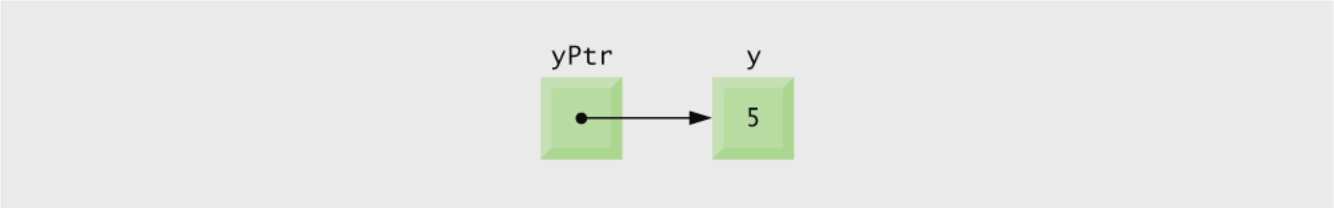
int a;

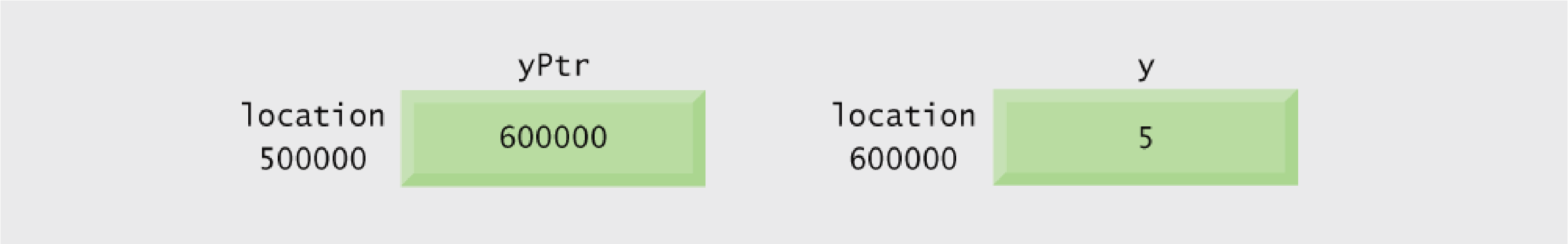
the address of a?



131

Graphical representation of a pointer pointing to an integer variable in memory.



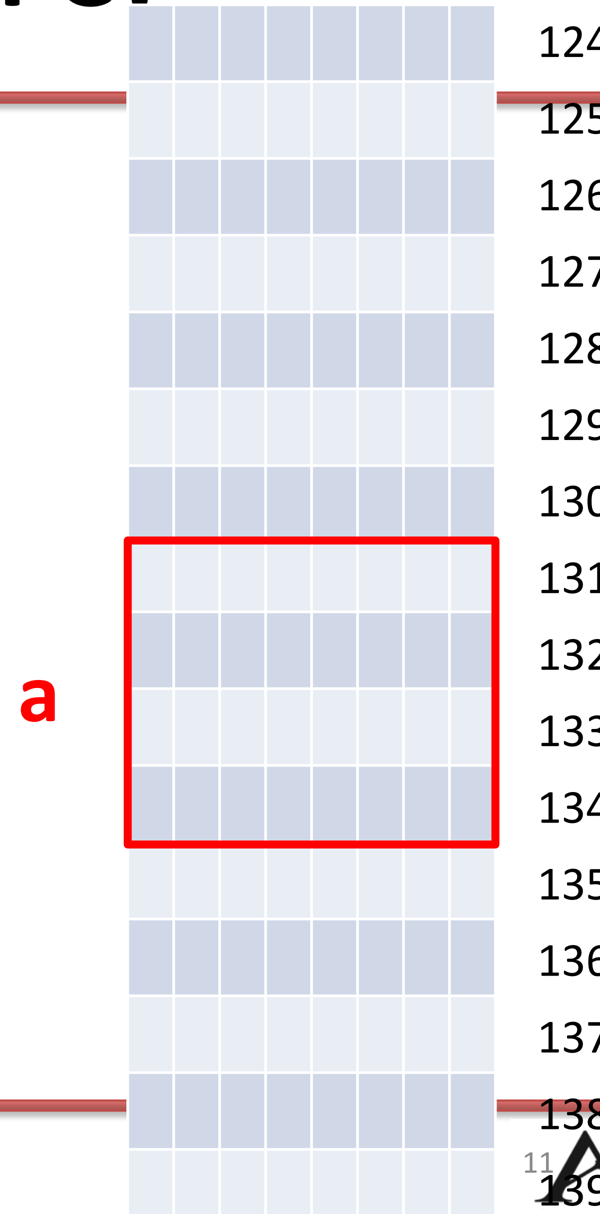


How to get addresses in C?

int a;

&a

Address of a!



Variable and its address

```
int  a = 10;
printf("a = %d and its address = %p\n", a, &a);
printf("sizes: %d and %d\n", sizeof(a), sizeof(&a));
```

- This is the output:

  a = 10 and its address = 0xbfefb304

  sizes: 4 and 4

- The output depends on the architecture!!
  - sizeof(a) → depends on the width of the memory
  - sizeof(&a) → depends on the length/size of the memory

Variable, addresses and pointers

```
int  a = 10;
int * b = &a;
printf("a = %d and its address = %p\n", a, b);
printf("sizes: %d and %d\n", sizeof(a), sizeof(b));
```

- The data type storing addresses are called pointers!
  - int * , char * , float * , double *

Pointers and changing what they point to

```c
int  a = 10;
int  c = 20;
int * b;
b = &a;
printf("b = %d and its address = %p\n", *b, b);
b = &c;
printf("b = %d and its address = %p\n", *b, b);
```

Addresses are data itself (actually, they are integers!)

b = 10 and its address = 0xbfdac9b4

b = 20 and its address = 0xbfdac9b0

Pointers

```
int a = 10;
int * b = &a;

*b = 20;

a = 2 / *b + 25;
```

- Initialization is important since a pointer initially points to an arbitrary memory position, which may not belong to your program!
- A good practice:
  - int *a = NULL;

Pointer arithmetic

```
int *a;
printf("a = %p  a+1 = %p", a, a+1);
a = 0xbff30330 a+1 = 0xbff30334

char *c;
printf("c = %p  c+1 = %p", c, c+1);
c = 0xbff30337 c+1 = 0xbff30338
```

So, the difference depends on the data type!

Pointer arithmetic

- Pointer arithmetic is independent of the data type.
- In other words, if p is a pointer, p+1 points to the next object (whether it is int, char, float or double).
- So, p+1 is not necessarily the next byte in the memory!!!

- So, then, how can we check the number of bytes between two pointers? Two options:

1. (p2 – p1) * sizeof(int)

   → for integers

2. ((int)p2 – (int)p1)

Pointer arithmetic

- We have the following defined for pointers as well:

int *a = &b;

a++, ++a

a--,   --a

- Pointer operators(&, *) has the same precedence with ++, -- (and unary +, -)!!
- They are right-to-left associative!!
  - *++cp → *(++cp)
  - *cp++ → *(cp++)

void and NULL pointers

- A pointer initially has an arbitrary value; i.e., it points to something arbitrary.
- Make it a habit to assign all pointers to NULL first.
  - int * a = NULL;

- void *
  → Generic pointer
  → Useful especially in cases where we don't know the type of the data beforehand!

Pointer Comparison

- Equality comparison is meaningful between:
  - ➢Pointers of the same type
  - ➢A pointer with a void pointer
  - ➢A pointer and a NULL pointer
- The result is true if the operands point to the same *object*

- For other relational operators (<, <=, >, >=):
  - – Result is based on the relative addresses of the objects pointed to.

Pointer Comparison: Example

```
if( p != NULL )   *p = 10;


char string[100];
char* p = &string[10];
if( p < &string[80] )
        printf("A");
if( p > string )
        printf("B");
```

Write the strlen() function using pointers.

Pointer Conversion

- similar to conversion of regular data types (i.e., int, float, double, char, ..), we can convert pointers:

```
int * a;
double *d;
a = (int *) d;
```

- While converting a pointer to a pointer of a bigger data type, you have to be cautious.
- In old architectures, you have to be careful while converting to small data types as well:
  - ip = (int *) cp;

Pointer & Strings

```
char * cp;
cp = “abc”;
```

- The following is possible:

```
char a = cp[0];
```

- If you do the following, you would get segmentation fault:

```
cp[0] = ‘A’;
```

Pointer & Strings

```
char * cp;
cp = “abc”;
cp[0] = ‘A’;

vs

char c[] = “abc”;
c[0] = ‘A’;
```

```
void f(char *c)
{
c[0] = ‘A’;
}

~~~~~~~~

f(“abc”); /* Seg. Fault */
```

Implement some string functions with pointers

- int strlen(char *sP);

- void strcpy(char *destP, char *sourceP);

Dynamic Memory Management

- sizeof() operator
- void * malloc(size_t size);
- void * calloc(size_t nobj, size_t size);
- void * realloc(void *p, size_t size);

Dynamic Memory Management

- void free(void *p);

Example

- Assume that you have numbers given on separate lines and that you do not know how many numbers there are in each line.
- The task is to read the numbers on each line and compute an average of the numbers on each line.
- Two cases:
  1. The number of lines & the number of numbers on each line is known.
  2. Neither the number of lines nor the number of numbers on each line is known.

Pointers & Functions

- Remember that functions are called in C by "call by value"?

- Now we can make "call by reference"

```
void f(int *N)
{
        *N = 10;
}
```

It is actually a "fake" call by reference.

Returning Multiple Values

- Now, we can return multiple values:

```
void f(int N, int *O, double *P)
{
        *O = N * N;
        *P = sqrt(N)
}
```

Examples

1. c = *++cp
2. c = *cp++
3. c = ++*cp
4. c = *--cp
5. c = *cp--
6. c = --*cp
7. c = (*cp)--
8. c = (*cp)++

Example

- Write the factorial function in a recursive way with the following declaration:

```
void fact(int N, int * result);
```

- Write the factorial function in a recursive way with the following declaration:

```
void fact(int *N);
```

Array vs. Pointer

- Array is basically a constant pointer

```
void g(int a[])
{
        printf("%d\n", a[0]);
}

void f(int * a)
{
        printf("%d\n", a[0]);
        g(a);
}
```

```
int main()
{
int b[] = {-1, 2, 3, 4};

        f(b);

return 0;
}
```

int a[] = {1, 2, 3};

int *b = a;

Array vs. Pointer

- Array is basically a constant pointer
- Hence, I can use a pointer like an array:

```
int *a;
~~~~~~~~~
x = a[2];
x = &(a+2);
```

Identical

Array vs. Pointer

- What does “an array is constant pointer” mean?
- It means that things like following are **not possible** (for the following definitions: int a[3]; int *b;):

```
a = b;
a++;
```

Array vs. pointer

- Arrays as function arguments

```
double avg(int aP[], int lengthP)
{
double sumL = 0.0;
int i;
    for(i=0; i < lengthP; i++)
        sumL += aP[i];

return sumL/lengthP;
}
```

v.s.

```
double avg(int * aP, int lengthP)
{
double sumL = 0.0;
int i;
    for(i=0; i < lengthP; i++)
        sumL += aP[i];

return sumL/lengthP;
}
```

Array vs. pointer

- Arrays as function arguments

```
double avg(int aP[], int lengthP)
{
double sumL = 0.0;
int i;
    for(i=0; i < lengthP; i++)
        sumL += aP[i];

return sumL/lengthP;
}
```

V.S.

```
double avg(int * aP, int lengthP)
{
double sumL = 0.0;
int * lastL = aP + lengthP;

    for( ; aP < lastL; aP++)
        sumL += *aP;

return sumL/lengthP;
}
```

Array vs. pointer

```
double avg(int aP[], int lengthP)
{
double sumL = 0.0;
int i;
    for(i=0; i < lengthP; i++)
        sumL += aP[i];

return sumL/lengthP;
}
```

```
double avg(int * aP, int lengthP)
{
double sumL = 0.0;
int * lastL = aP + lengthP;

    for( ; aP < lastL; aP++)
        sumL += *aP;

return sumL/lengthP;
}
```

For either of the above definitions, the following are valid:

```
int a[] = {1, 2, 3, 4, 5};
avg(a, 5);
avg(&a[2], 3);
avg(a+2, 3);
```

Multi-dimensional Arrays & Pointers

- Compare the following:

```
int a[M][N];

int *a[N];

int (*a)[N];
```

Pointer to Pointers

- Compare the following (there is no limit on the level of ‘pointing’ to pointers):

```
int *a;

int **b;

int ***c;
```

Pointers to Functions

- Pointer to function
  - Contains address of function
  - Similar to how array name is address of first element
  - Function name is starting address of code that defines function
- Function pointers can be
  - Passed to functions
  - Stored in arrays
  - Assigned to other function pointers

Pointers to Functions

- Example: bubblesort
  - Function `bubble` takes a function pointer
    - `bubble` calls this helper function
    - this determines ascending or descending sorting
  - The argument in `bubblesort` for the function pointer:
    - `int ( *compare )( int a, int b )`

    tells `bubblesort` to expect a pointer to a function that takes two `int`s and returns an `int`
  - If the parentheses were left out:
    - `int *compare( int a, int b )`
    - Defines a function that receives two integers and returns a pointer to a `int`