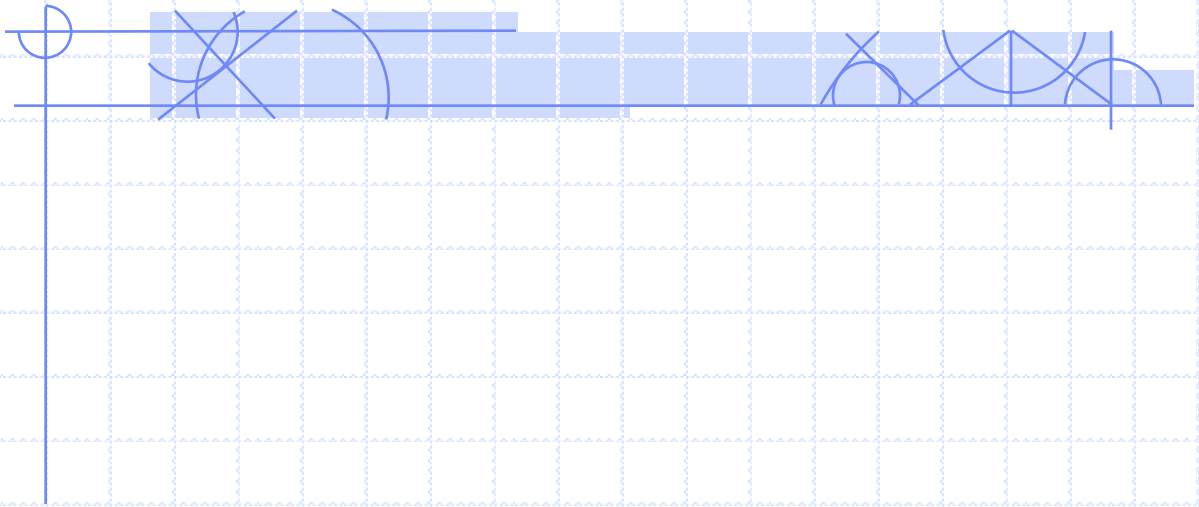
Pointers



Variable Address

- A variable stores a value represented by the variable's name, or identifier

```
int i = 5;  i++; printf("%d", i);
```

- The variable value inhabits certain number of memory cells
- The variable's name is associated with the location of these memory cells
- The memory location is specified by memory address
- The address is considered as a hexadecimal number

Pointer Variable

- A pointer is a variable whose value is the address of some other object
- The address of an object is obtained by & operator, like in

  scanf("%d", &x);
- The object can have any valid type: int, float, struct, etc.
- The pointer variable is declared by identifier, and by type.

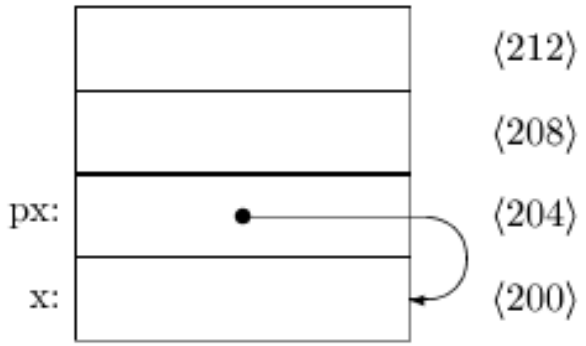
  <type> *<identifier>
- The type of the pointer is the type of the objects whose addresses this pointer can hold
- The identifier is the name of the pointer variable
- A pointer variable is stored in one machine word, regardless of the variable type to which it points

Example

- If a pointer to an integer is declared as

  int *px;

  - px is the name of the pointer;
  - px is a variable of type int *;
  - int * means a pointer to an integer;
  - the value *px is of type integer.



Reference and De-reference

- Since a pointer variable contains the address of another variable, the value of that variable can be found at that address:

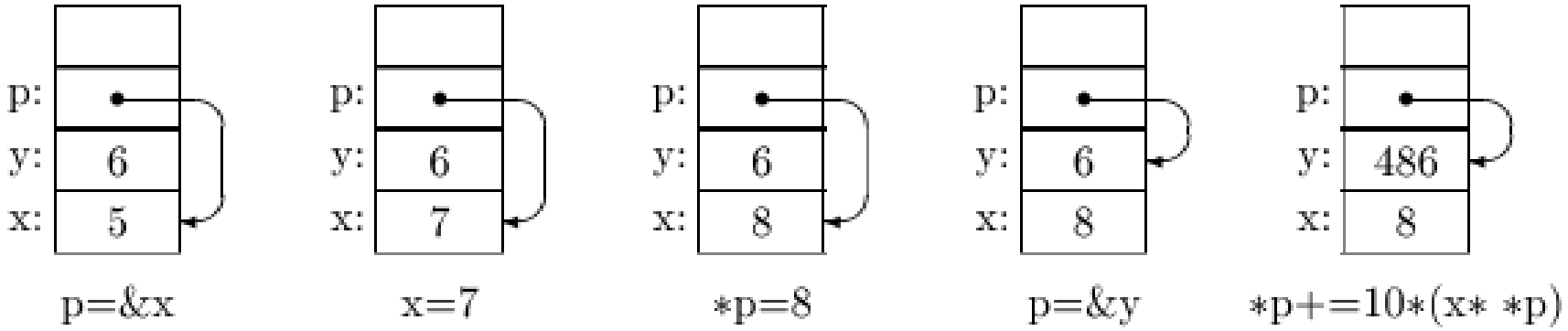
```
main()
{
    int x, *px;          /* defines the pointer px */
    px = &x;             /* px assigns &x => address of x */
    *px = x;             /* *px => x is the value px points to */
}
```

Example Program

```
main()
{
    int x=5, y=6, *p;

    p = &x;                 /** pointer needs to point to something **/
                    printf("1. x=%d, y=%d, *p=%d\n", x, y, *p);
    x = 7;          printf("2. x=%d, y=%d, *p=%d\n", x, y, *p);
    *p = 8;         printf("3. x=%d, y=%d, *p=%d\n", x, y, *p);
    p = &y;         printf("4. x=%d, y=%d, *p=%d\n", x, y, *p);
    *p += 10 * (x * *p);
                    printf("5. x=%d, y=%d, *p=%d\n", x, y, *p);
}
```

Output



1. x=5, y=6, *p=5
2. x=7, y=6, *p=7
3. x=8, y=6, *p=8
4. x=8, y=6, *p=6
5. x=8, y=486, *p=486

Pointer Operations

- Standard operations
  - Assignment
    - an address of a variable of any type, or of an array element
    - a NULL constant - the pointer points to nowhere
    - the address of a variable which is a pointer itself

Examples

```
int *p, x;
void *v:
v = &x;                 /* address of a variable */
p = NULL;               /* NULL address */
p = (int *) 1778;       /* an absolute address */
v = p;
```

- Pointers can not point to constants, ordinary expressions or register variables.

Pointer Operations

- Specific pointer arithmetic
  - adding an integer constant to a pointer (including increment)
  - subtracting an integer constant from a pointer (including decrement)
  - subtracting one pointer from another of the same type
  - comparing two pointers of the same type

Examples

```
#define NULL 0
main()
{
    int x, y;
    int *px=(&x);          /** can initialize an automatic pointer **/
    int *py;
    void *pv;
    py = px;               /** assign to pointer of same type **/
    px = (int *) pv;       /** recast a (void *) pointer **/
    pv = (void *) px;      /** recast to type (void *) **/
    py = px+2;             /** for use with arrays **/
    px++;                  /** for use with arrays **/
    if (px == NULL) ...    /** compare to null pointer **/
    py=NULL;               /** assign to null pointer **/
}
```

Invalid Pointer Operations

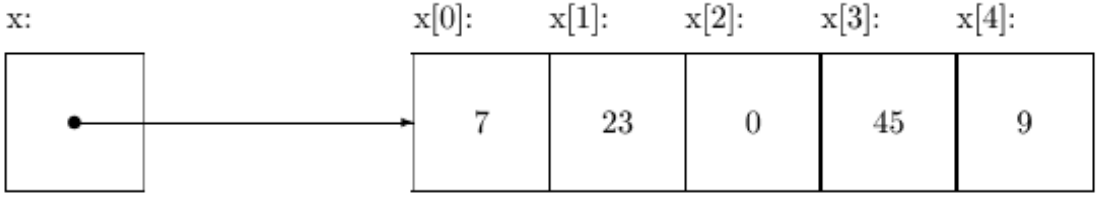
- Adding two pointers
- Multiply and divide pointers
- Shift or mask pointers
- Add float or double numbers to a pointer
- Assign pointers of different types without a cast

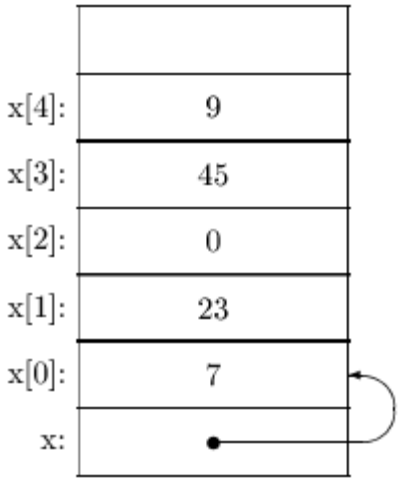
Wrong Examples

```
/** Illegal Pointer Operations **/
main()
{
    int x, y;
    int *px, *py, *p;
    float *pf;
    px = &x;            /** legal assignment **/
    py = &y;            /** legal assignment **/
    p = px + py;        /** addition is illegal **/
    p = px * py;        /** multiplication is illegal **/
    p = px + 10.0;      /** addition of float is illegal **/
    pf = px;            /** assignment of different types is illegal **/
}
```

Pointers and Arrays

- An array is a contiguous space in memory which holds a certain number of objects of one type
- It is enough to know the address of the first cell in order to locate the others
- The array name is considered as a pointer to the array elements
- The array and the pointer are closely related, in that
  - &x[i] and (x+i) are both pointers to the i+1 element
  - x[i] and *(x+i) both get the i+1 element in the array
- The array notation is a form of relative addressing, elements is absolute addressing





Pointers and Arrays

- There are differences between pointers and arrays
  - array name is a constant, while a pointer is a variable, so

```
int x[10], *px;
px = x; px++;   /** valid **/
x = px; x++;    /** invalid, cannot assign a new value **/
```

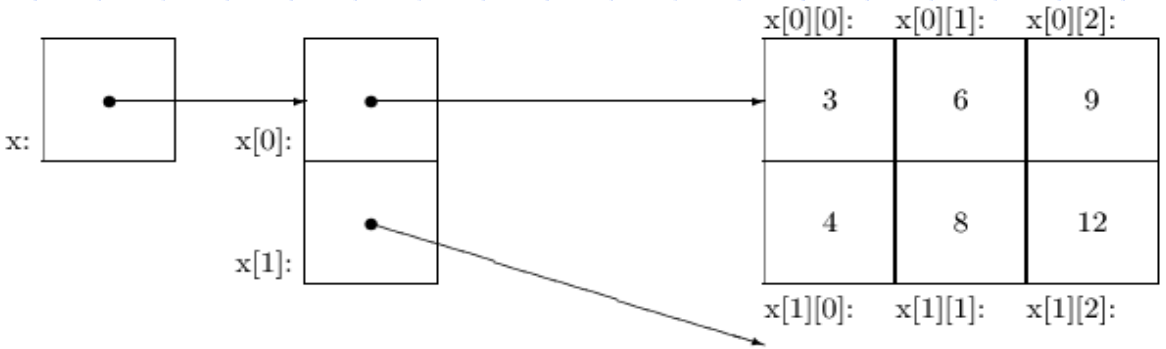
  - defining the pointer only allocates memory space for the address, and the pointer does not point to anything, while defining an array (int x[10];) gives a pointer to a specific place in memory and allocates enough space to hold all the array elements
- To allocate space for an array declared as a pointer, use
  - *malloc() or *calloc(), functions declared in <stdlib.h>
  - free() to deallocate the space after has been used.

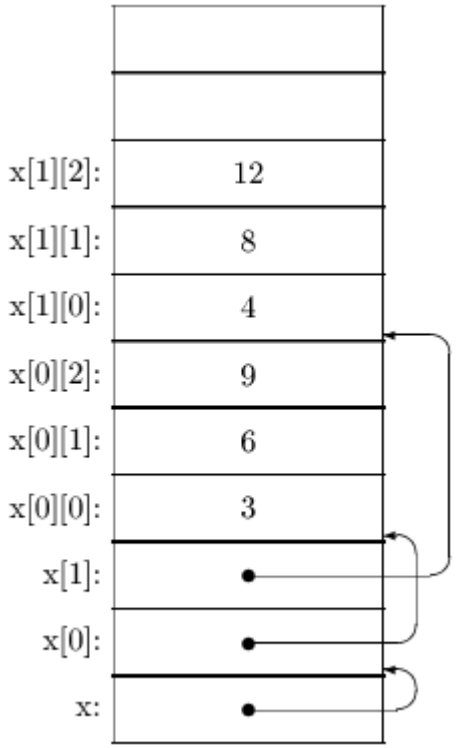
Memory Allocation

```
/*** memory_allocation for arrays referenced by pointers ***/
#include <stdlib.h>
main()
{
        int n;
        float *a, *b;
        a = (float *) malloc(n * sizeof(float));        /** not initialized **/
        b = (float *) calloc(n,  sizeof(float));        /** initialized to 0 **/
        if ((a == NULL) || (b == NULL))
        printf("unable to allocate space");
        ...
        ...
        free(a);
        free(b);
}
```

Multidimensional Arrays

- Array of arrays
  - everyone introduces a dimension (index)
  - the elements can be referenced relatively or absolutely





Multidimensional Arrays

- A single element of a two-dimensional array can be addressed at least in four different ways by implementation of either relative or absolute, or both referencing:

  td[1][3]
  (td[1] + 3)
  (td + 1)[3]
  ((td +1) + 3)

- A pointer to a pointer can be used instead of a multidimensional array

  int **y;

  - y points to a one-dimensional array whose elements are pointers, *y points to the first row, and **y is the first value.

- When passing a two-dimensional array to a function, it can be referenced in four ways:
  - **y, as a pointer to a pointer;
  - *y[], an array of pointers;
  - y[][COL], an array of arrays, unspecified number of rows; or
  - y[ROW][COL], an array of arrays, with the size fully specified.

Pointers and Functions

- Pointers as Function Arguments - Call by Value
  - when a function is called, it gets a local copy of the arguments
    - it cannot affect a value passed to it, only its local copy
  - if it is necessary to change the original values, not the arguments, but their addresses can be passed instead - Call by Reference
    - the function gets a copy of the address, which gives it access to the value itself
    - the function can affect the global values
- For example, to swap two numbers:

Example: swap two numbers

```
main()
{    float x=5.0, y=6.0;
       void swap_A(float *x, float *y), swap_V(float x, float y);
       printf("x = %6.2f, y = %6.2f\n", x, y);
       swap_V(x, y);                printf("x = %6.2f, y = %6.2f\n", x, y);
       swap_A(&x, &y);              printf("x = %6.2f, y = %6.2f\n", x, y);
}
void swap_V(float x, float y)                /** passes values of x and y **/
{
       float tmp = x;
       x = y;
       y = tmp;
}
void swap_A(float *x, float *y)              /** passes addresses of x and y **/
{
       float tmp = *x;
       *x = *y;
       *y = tmp;
}
```

Pointers and Functions

- Functions Returning Pointers
  - a pointer can be returned by a function
  - three pointers must be of the same type (or be recast appropriately):
    - the function return-type
    - the pointer type in the return statement
    - the variable the return-value is assigned to
  - the pointer should not point to an automatic local variable within the function, since the variable will not be defined after the function is exited so the pointer value will be invalid

Example - I

```
/*** returns a pointer to the maximum value in an array ***/
int maximum_val1(int [], int);
int maximum_val2(int *, int);
int *maximum_ptr1(int *, int);
int *maximum_ptr2(int *, int);
main()
{    int i,n;
     int A[100], *max;
     printf("number of elements: "); scanf("%d",&n);
     printf("input %d values:\n", n);
     for (i=0; i<n; i++) scanf("%d", A+i);
     printf("maximum value = %d\n", maximum_val1(A,n));
     printf("maximum value = %d\n", maximum_val2(A,n));
     max = maximum_ptr1(A,n);
     printf("maximum value = %d\n", *max);
}
```

Example - II

```
int maximum_val1(int A[], int n)
{    int max, i;
     for (i=0, max=0; i<n; i++)
          if (A[i] > max) max = A[i];
     return max;
}

int maximum_val2(int *a, int n)
{    int max=0;
     for ( ; n>0 ; n--, a++)
          if (*a > max) max = *a;
     return max;
}
```

Example - III

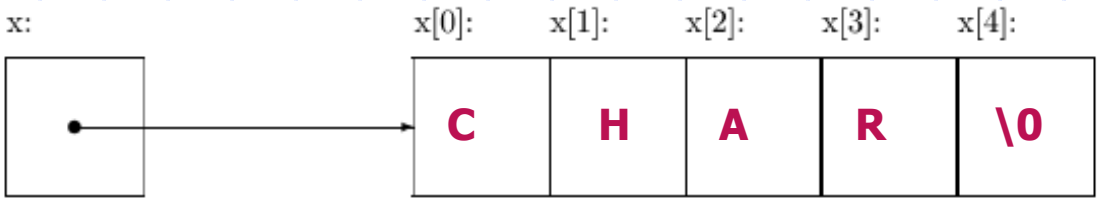
```
int *maximum_ptr1(int *a, int n)                    /*** will work ***/
{    int *max = a;
     for ( ; n>0; n--, a++)
         if (*a > *max) max = a;
     return max;                /*** max points a cell in the array ***/
}

int *maximum_ptr2(int *a, int n)                    /*** won't work ***/
{    int max = *a;
     for ( ; n>0; n--, a++)
         if (*a > max) max = *a;
     return &max; /*** max will not exist after function ends ***/
}
```

Strings

- Strings are represented as one-dimensional arrays of type char
- In the memory strings are stored in a sequence of one-or-two byte locations
- The sequence is always completed by a character '\0'
- Example

  char x[5] = { 'C', 'H', 'A', 'R' };



Strings

- Strings can be represented by a pointer of type char to reference its characters
  - a pointer to the firs character only is needed
  - all the other characters can be found at the following addresses
  - the last one will be just before the character '\0'
- Example

```
char *p;
p = "Good day";
printf("%s\n", p);
printf("%s\n", p+5);
```

- The character 'd' only can be referenced as *(p+5), as well as "Good day"[5].
- The character '\0' is not printed, it is only used to indicate the end of the sequence to be printed or processed

String Objects

- String constant
  - " some text"
- String variable
  - its size is needed for declaration
    - given explicitly
    - especially allocated – dynamically created
  - The function malloc() can be used
    - parameter of malloc() is the number of bytes to be allocated
    - result is a pointer to the allocated memory
    - prototype of this function is in <alloc.h>
- Example:

```
#include <alloc.h>
char *s;
s = malloc (100);
```

- The string characters can be accessed and modified by incrementing the pointer which defines the string
- For protection of a string against modification, a const modifier can apply
- Example:

```
char s1[] = "Day";
const char *s2 = "Night";
```

Example

```
// Use of pointers with the string copy
main()
{
      char *t = "hello", s[100];
      void strcpy(char *, char *);
      strcpy(s,t);
      printf("%s\n", s);                  /** will print 'hello' **/
}
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)                   /** while ((*s++ = *t++) != '\0') **/
;
}
```