ITC213: STRUCTURED PROGRAMMING

Bhaskar Shrestha
National College of Computer Studies
Tribhuvan University

Lecture 11: Pointers

Readings: Chapter 10

Introduction (1/2)

- The correct understanding and use of pointers is crucial to successful C programming
- Reasons
  - Pointers provide the means by which functions can modify their calling arguments
  - Pointers provides the necessary support for dynamic memory allocation
  - Pointers can improve the efficiency of certain routine involving arrays
  - Pointers provide support for dynamic data structures, such as linked lists and binary trees

Introduction (2/2)

- Pointers
  - A variable that holds a memory address
  - Powerful, but difficult to master
  - Simulate call-by-reference
  - Close relationship with arrays and strings

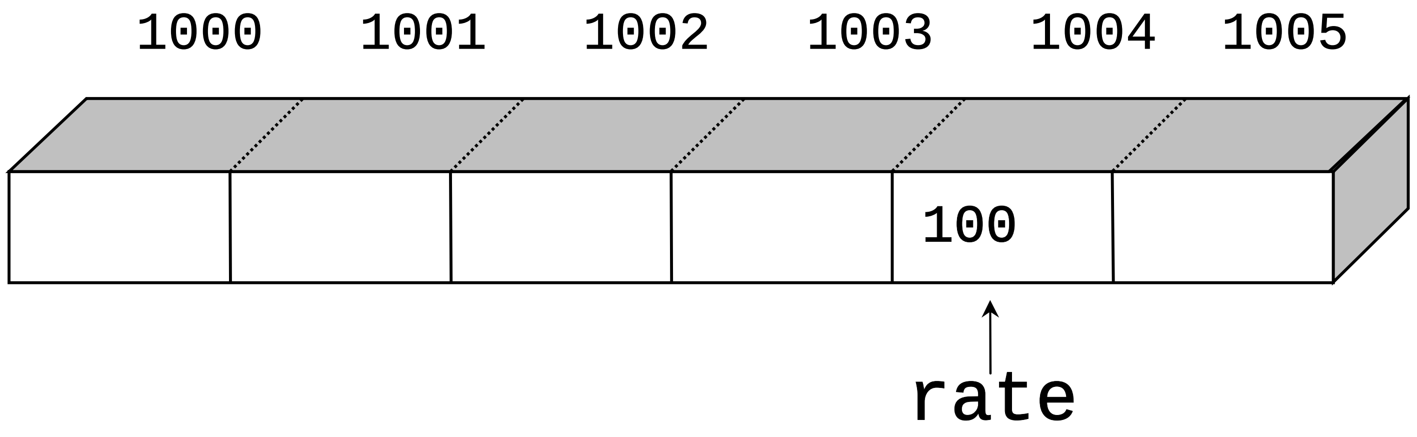
The Computer’s Memory (1/2)

- PC’s RAM consists of many thousands of contiguous memory locations, and each location is identified by a unique address
- The memory addresses range from 0 to a maximum value
- Operating system uses some of the system’s memory
- A program while running uses some of the memory for the program code and data

The Computer’s Memory (2/2)

- Amount of memory required to store a data depends on the type of data
- For each variable declared,
  - compiler sets aside a memory location
  - the address is unique
  - compiler associate’s the variable name with the address
  - proper memory location is automatically accessed when you refer a variable

A variable in memory



A variable rate stored at a specific memory address 1004 with value 100

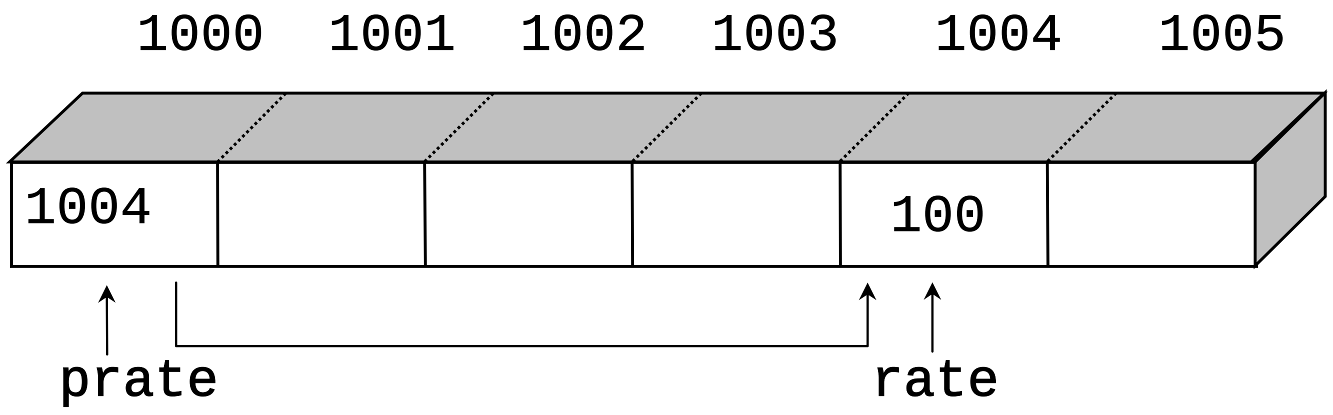
Creating a Pointer

- Address of a variable is a number and can be treated like any other number in C
- If you know a variable’s address, you can access the value stored there
  - The address of a variable can accessed using the `&` operator (for e.g., `&rate`)
- For this, you can create a second variable in which to store the address of the first
- This type of variable is called a pointer variable

Creating a Pointer

- Creating a pointer
  - declare a variable to hold the address of `rate` and name it `prate`
  - store the address of the variable `rate` in the variable `prate`
    - `prate = &rate;`
  - Now, `prate` points to `rate`, or is a pointer to `rate`
- ***A pointer is a variable that stores the memory address , possibly, of another variable***

A pointer variable in memory



The variable prate contains the address of the variable rate and is therefore a pointer to rate

Declaring Pointer Variables

- `type *ptrname;` (with *type* in italics)
- where *`type`* is the base type of the pointer and may be any valid data type
- The name of the pointer variable is specified by `ptrname`
- The asterisk (`*`) is the ***indirection*** operator, and it indicates that `ptrname` is a pointer to type *`type`* and not a variable of type *`type`*

```
char *ch1, *ch2;

float *value, percent;
```

`ch1` and `ch2` both are pointers to type `char`
`value` is a pointer to type `float`, and `percent` is an ***ordinary*** `float` variable

Initializing Pointers

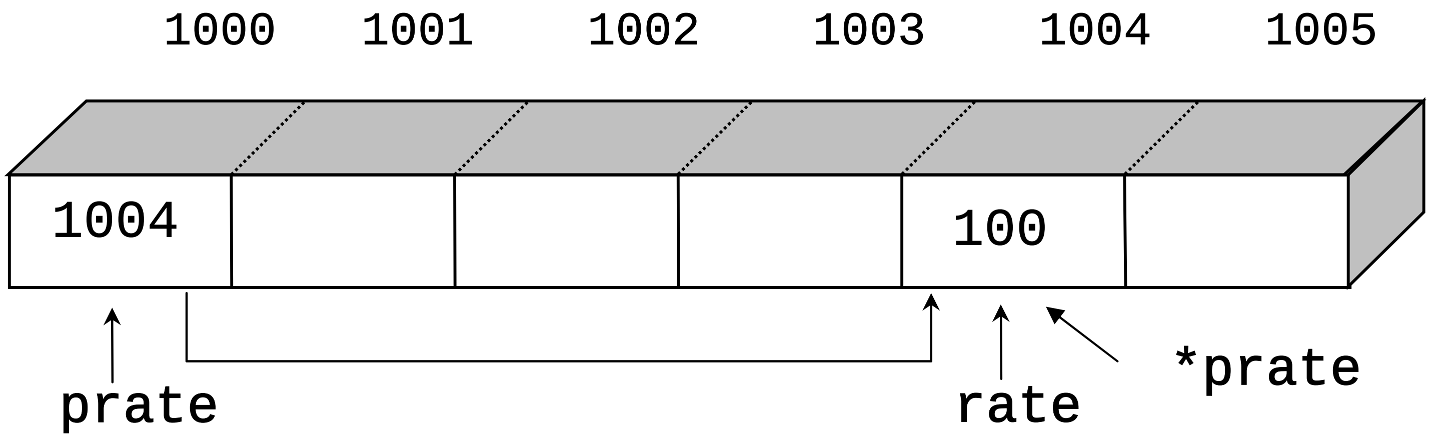
- You cannot do anything until you assign a pointer variable a valid address
- **Uninitialized pointers are disastrous**
- Use the ***address-of*** **`&`** operator to get the address of a variable and assign it to a pointer variable

  -
    ```
    int rate, *prate;
    prate = &rate;
    ```
  - `prate` gets the address of `rate`, `prate` points to `rate`
- `&` is a unary operator, which returns the address of its operand

The Indirection Operator (Getting Pointed to value)

- When the ***indirection*** operator ***** precedes the name of a pointer variable, it refers the value of the variable pointed to
- Writing
  - `printf("%d", rate);`
    is same as `printf("%d", *prate);`
- Accessing the contents of a variable by using the variable name is called *direct access* (reference)
- Accessing the contents of a variable by using a pointer to the variable is called *indirect access* (reference) or *indirection*

The Indirection Operator



Use of the indirection operator with pointers

Basic Pointer Usage

```
int rate = 100;
int *prate; /* Declare a pointer to int */

prate = &rate; /* initialize prate to point to rate */

/* Access rate directly and indirectly */
printf("Direct access, rate = %d\n", rate);
printf("Indirect access, rate = %d\n", *prate);

/* Display the address of rate two ways */
printf("\nThe address of rate = %p\n", &rate);
printf("The address of rate = %p\n", prate);
```

Use %p to print address in the format used by the host OS

Use of *

- `*` is a unary operator and can only be used with pointer variables
- If `prate` points to `rate` (i.e., `prate = &rate`), then an expression such as `*prate` can be used interchangeably with its corresponding variable `rate`
- Thus `*` can be used for assignment
  - `*prate = 200;`
    Indirectly changes the value of `rate` to 200

Use of Indirection Operator

```
int v = 100;
int *pv;
int u1, u2, u3;

u1 = 2 * (v + 5);   /* ordinary expression */

pv = &v;            /* make pv point to v */
u2 = 2 * (*pv + 5); /* equivalent expression */

*pv = u1;           /* change value of v to u1(210) */
u3 = 2 * (*pv + 5);

printf("v = %d, *pv = %d\n", v, *pv);
printf("u1 = %d, u2 = %d, u3 = %d\n", u1, u2, u3);
```

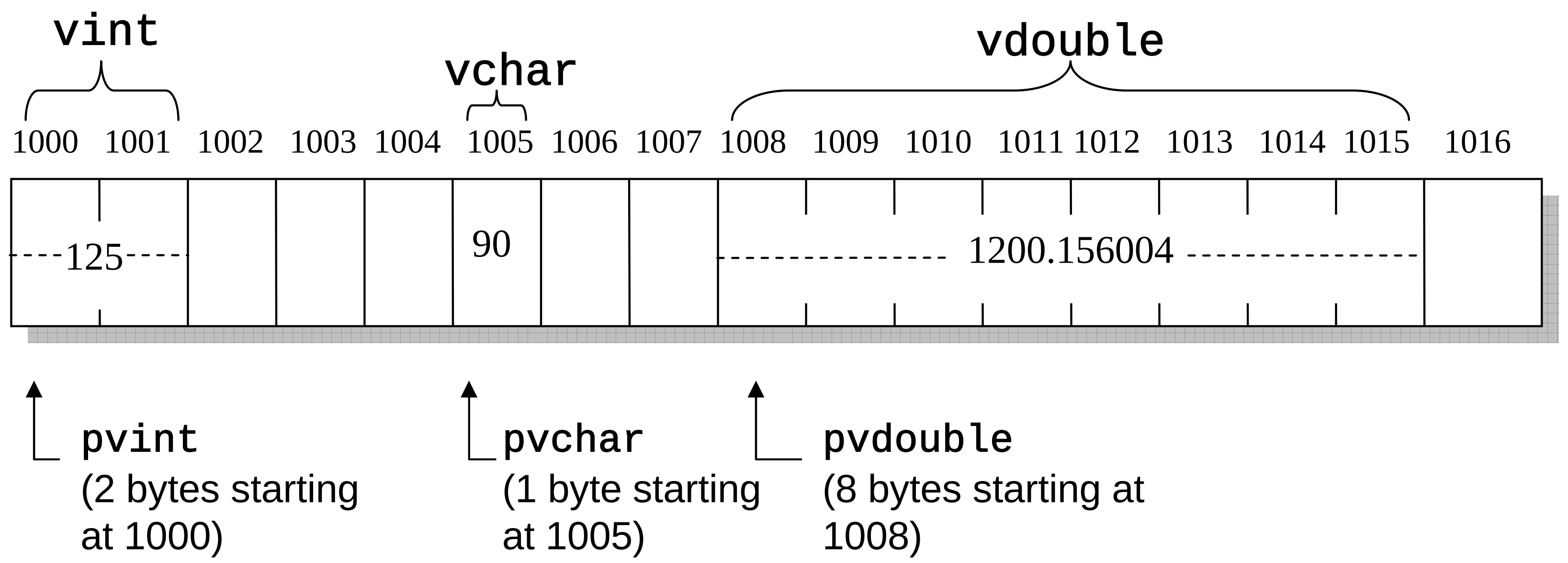
Quiz

- Given
  - int v = 100;
    int *pv = &v;
- What is the value of v after
  - `++ *pv;` **101, *pv is incremented**
    and `*pv ++;` **No change, pv is incremented, unary operators are evaluated from right to left**
- and what is the value of
  - &*pv; **The address of v (&v == pv)**
  - *&v; **The value of v (i.e., 100), * and & cancel each other**

Pointers and Variable Types

- Different variable types occupy different amounts of memory
  - `char` takes 1 byte, `int` takes 2 bytes and `float` take 4 bytes
- How pointers handle the addresses of multi-byte variables?
  - The address of a variable is actually the address of the first (lowest) byte it occupies

Pointers and Variable Types



```
int *pvint = &vint;

char *pvchar = &vchar;

double *pvdouble = &vdouble;
```

Each pointer is equal to the address of the first byte of the pointed-to variable. Thus, pvint equals 1000, pvchar equals 1005, and pvdouble equals 1008

The Pass by Value Mechanism

- C passes arguments to functions by value
- Recall in a call by value mechanism, if a parameter is altered within a function, the alteration of parameter will not alter its corresponding argument in the function call
- Consider the function call
  - `swap(a, b);`
- Because of call by value, `swap` can't affect the arguments `a` and `b` in the routine that called it The function on the left swaps copies of `a` and `b`

```
void swap(int x, int y)
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

Pass by Reference

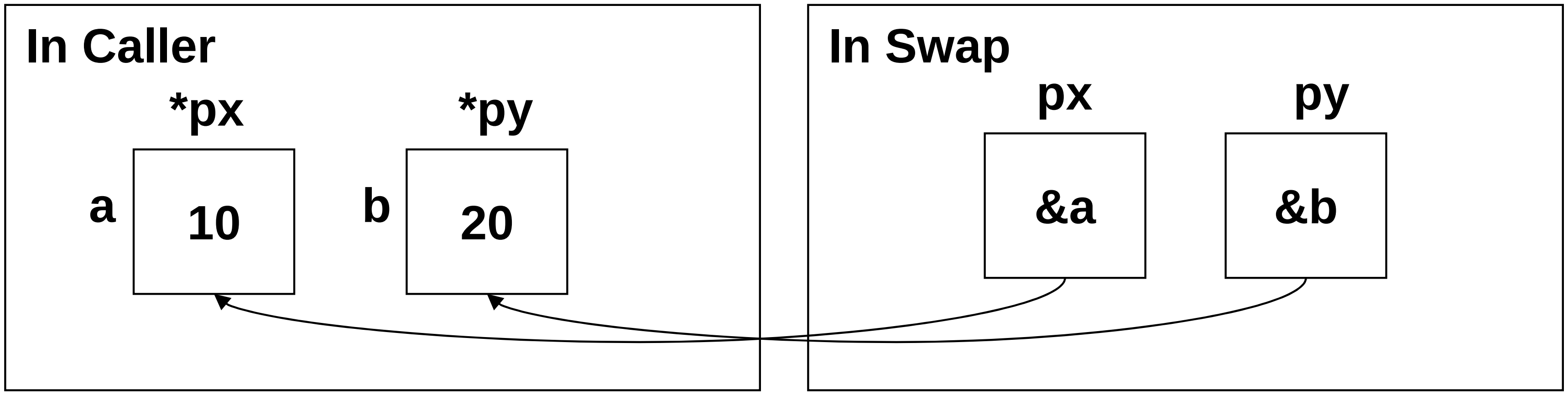
- To create a pass by reference
  - you need to pass the address of the variable
    - swap(&a, &b);
  - create a pointer variable in the formal parameter to receive the address passed
  - use the pointer variable in the function to modify the original argument

```
void swap(int *px, int *py)
{
   int temp;
   temp = *px;
   *px = *py;
   *py = temp;
}
```

Pass by Reference

```
int a=10, b=20;
swap(&a, &b);
printf(“%d %d”, a, b);
```

```
void swap(int *px, int *py)
{
   int temp;
   temp = *px;
   *px = *py;
   *py = temp;
}
```



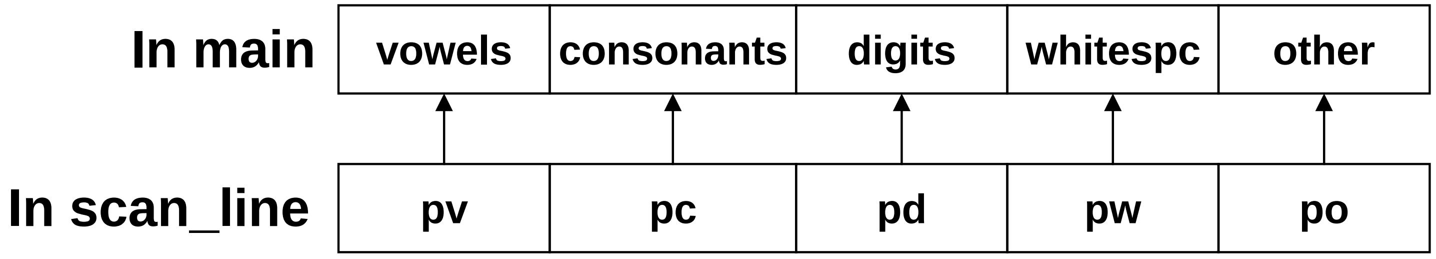
Another Example

- **Q**: Write a function that given a string ***s***, returns the no. of vowels, consonants, digits, whitespaces and other characters in it
- **Ans**:
  - The function needs to return multiple values
  - Recall, a function can't return multiple values
  - You can use pointer arguments to return multiple values
  - For this function, we have 6 arguments, 5 for count and 1 for the string ***s***

```
void scan_line(char [], int *, int *, int *, int *, int *);

main()
{
  char line [80];
  int vowels = 0, consonants = 0, digits = 0, whitespc = 0, other = 0;
  printf("Enter a line of text below:\n");
  scanf(" %[^\n]", line);
  scan_line(line, &vowels, &consonants, &digits, &whitespc, &other);
  /* now print the value of each counter */
}


void scan_line(char s[], int *pv, int *pc, int *pd, int *pw, int *po)
{
  char c; /* uppercase character */
  int count = 0; /* character counter */
  while ((c = toupper(s[count])) != '\0') {
    if(c =='A' || c =='E' || c =='I' || c =='O' || c =='U') ++ *pv;
    else if (c >= 'A' && c <= 'Z')  ++ *pc;
    else if (c >='0' && c <= '9')  ++ *pd;
    else if (c == ' ' || c == '\t')  ++ *pw;
    else  ++ *po;
    ++count;
   }
}
```



Passing Arrays to Functions Revisited (1/2)

- Recall
  - an array name is actually a pointer to the first element of the array
  - the array name represents the address of the first element in the array
  - Therefore, an array name is treated as a pointer when it is passed to a function
- An array name that appears as a formal parameter within a function definition can be declared either as a pointer or as an array of unspecified size
- The formal parameter is actually a pointer rather than an array

Passing Arrays to Functions Revisited (2/2)

- For example, the following program fragment passes the address of x to func1():

-
  ```
  int main()
  {
      int x[10];
      func1(x);
      /* ... */
  }
  ```
- Hence, to receive x, a function called func1() can be declared as

```
void func1(int *x)
{
      /* ... */
}
```

OR

```
void func1(int x[])
{
      /* ... */
}
```

- Both specify x as an integer pointer. First actually uses a pointer. The second simply that an array of type int of some length is to be received

Returning Pointers from Functions

- You can return a pointer, i.e. the address of some variable, from a function
- To return a pointer, a function must be declared as having a pointer return type

Scan returns a double pointer,
So the return type `double *`

```
double *scan(double []);

main()
{
  double x[100];
  double *p;

  p = scan(x);
}
```

```
double *scan(double z[])
{
  double *pf;
  ...
  /* process elements of z */
  pf = ....;
  return pf;
}
```

Using the const Qualifier with Pointers

- `const` qualifier: variable cannot be changed
  - Use `const` if function does not need to change a variable
  - Attempting to change a `const` variable produces an error
- `const` pointers
  - Point to a constant memory location
  - Must be initialized when declared
  - `int *const myPtr = &x;`
    - Type `int *const` – constant pointer to an `int`
  - `const int *myPtr = &x;`
    - Regular pointer to a `const int`
  - `const int *const Ptr = &x;`
    - `const` pointer to a `const int`
    - `x` can be changed, but not `*Ptr`

Pointer Expressions

- In general, expressions involving pointers conform to the same rules as other expressions
- We’ll see pointer expressions such as
  - Assignments of one pointer variable to another
  - Conversions between different types of pointer variables, and
  - Arithmetic operations on pointer variables
  - Pointer Comparison

Pointer Assignments

- You can use a pointer on the right-hand side of an assignment statement to assign its value to another pointer

  -
    ```
    int x = 99;
    int *p1, *p2;
    p1 = &x;
    p2 = p1;
    ```
  - Now both `p1` and `p2` point to `x`, you can use both `p1` and `p2` to access `x`

Pointer Assignments

```
int x = 99, y = 100;
int *p1, *p2;

p1 = &x;
p2 = p1;

/* print the value of x twice */
printf("Value at p1 and p2: %d %d\n", *p1, *p2);

/* print the address of x twice */
printf("Addresses pointed to by p1 and p2: %p %p\n",
                                 p1, p2);

p2 = &y; /* now make p2 point to y */
printf("Value at p1 and p2: %d %d\n", *p1, *p2);
```

Pointer Conversions

- One type of pointer can be converted to another
- Use a cast to convert a pointer from one type to another

```
int x=258;
char *p;
p = (char *) &x;
printf("The first byte of x is %d\n", *p);
p++;
printf("The second byte of x is %d\n", *p);
```

Increments p, more about this later

The void *

- Conversion between any type of pointer and void * does not require a cast
- void * is a generic pointer
  - Used to specify a pointer whose base type is unknown
  - Allows a function to specify a parameter that is capable of receiving any type of pointer argument without reporting a type mismatch
  - Used to refer to raw memory (such as that returned by the malloc() function)
  - Cannot dereference a void *

```
int x = 100, y;
void *p;
p = &x; /* no need of cast */
y = *(int *)p; /* can’t apply * directly to p */
printf("value of y is %d\n", y);
```

Pointer Conversion Problem

```
double x = 100.1, y;
int *p;

/* the next statement cause p (which is an
      integer pointer) to point to a double. */
p = (int *) &x;

/* the next statement does not operate as expected. */
y = *p; /* attempt to assign y the value x through p */

/* the following statement won't output 100.1 */
printf("The (incorrect) value of x is %f\n", y);
```

Pointer operations are performed relative to the base type of the pointer not by the type pointed to.

The NULL Pointer

- In general, it does not make sense to assign an integer value to a pointer variable
- However, you can assign a pointer variable the value 0 (without a cast)
  - Which means the pointer variable is currently not pointing to any variable
- The symbolic constant NULL is used in placed of 0

-
  ```
  #define NULL 0
  float u, v;
  float *pv = NULL; /* pv is not pointing to
  any varible, so *pv should not be used */
  ```

Pointer Arithmetic

- You can
  - Add an integer to a pointer
  - Subtract an integer from a pointer
  - Subtract two pointers of same type
- **The above operations are meaningless unless the pointer variable points to an array element**
- Besides these no other operations are allowed

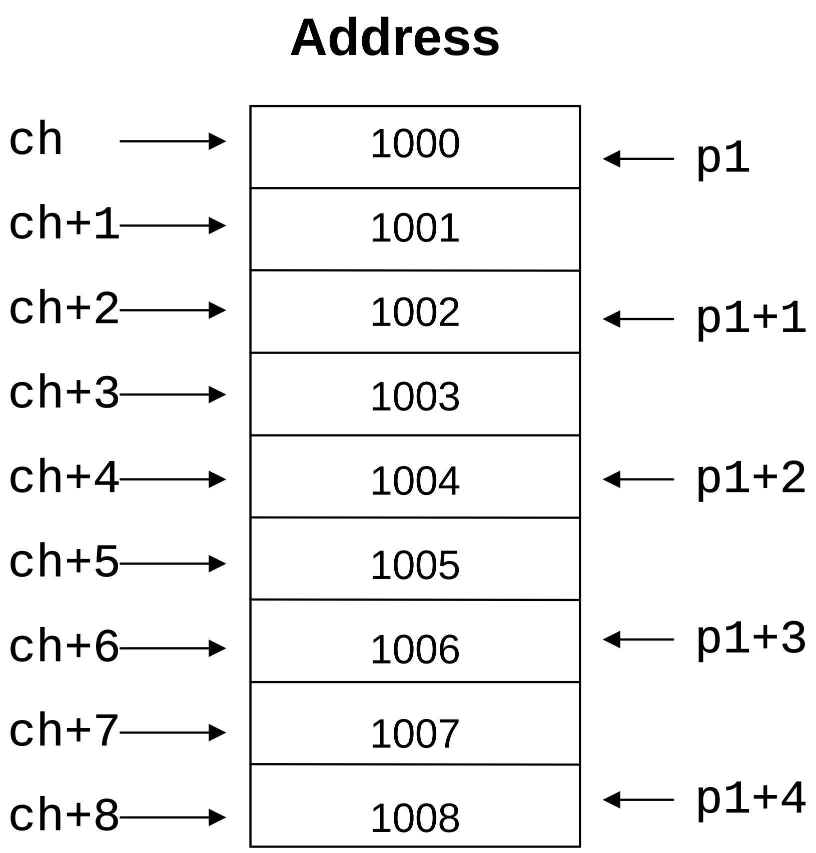
Pointer Addition

- Let `p1` be an integer pointer with the current value of 1000. Also, assume `int`s are 2 bytes long
- After the expression
  - `p1++;`
    `p1` contains 1002, not 1001
- Each time `p1` is incremented, it will point to the next integer variable after 1000
- Hence, the expression
  - `p1--;`
    causes `p1` to have the value 998
  - `p1 = p1 + 12;`
    makes `p1` point to the 12th element of p1's type beyond the one it currently points to

Pointer Addition

- All pointer arithmetic is relative to its base type (assuming 2-byte integers)
- Pointer arithmetic is used only when a pointer variable points to an array element
- In such case, the same pointer variable can be made to point to any element of the array

```
char *ch = (char *)1000;
int *p1 = (int *) 1000;
```



```
int array[5] = {5,10,15,20,25};
int *p1;

p1 = &array[0]; /* you can also write p1 = array; */
printf("First element of the array --> ");
printf("Address: %p, value: %d\n", p1, *p1);

p1++;
printf("Second element of the array: --> ");
printf("Address: %p, value: %d\n", p1, *p1);

p1 = p1 + 3; /* you can also write p1 += 3; */
printf("Fifth element of the array: --> ");
printf("Address: %p, value: %d\n", p1, *p1);

p1--;
printf("Fourth element of the array: --> ");
printf("Address: %p, value: %d\n", p1, *p1);

p1 = p1 - 3;
printf("First element of the array --> ");
printf("Address: %p, value: %d\n", p1, *p1);
```

Subtracting Pointers

- You can subtract one pointer from another in order to find the number of objects of their base type that separate the two
- The two pointers must be of same type
- Use if the two pointers point to different elements of same array

```
int *px, *py;
int a[6] = {1, 2, 3, 4, 5, 6};

px = &a[0];
py = &a[5];
printf("px = %p, py = %p", px, py);
printf("\n\npy - px = %d\n", py - px); /* prints 5 */
```

Pointer Comparisons

- Pointer variables can be compared provided both variables are of the same data type
  - Useful when both pointer variables point to elements of the same array
  - Can test for either equality or inequality
  - Can be compared with zero
- Let `px` and `py` point to elements of same array
  - `(px < py)`
    is `px` pointing to an *element ahead* of `py`?
  - `(px >= py)`
    is `px` pointing to an *element after or to same as* `py`?
  - **What do `(px == py)`, `(px != py)` and `(px == NULL)` mean ?**

Relationship between an Array and a Pointer (1/3)

- Arrays and pointers closely related
  - Array name like a constant pointer
  - Pointers can do array subscripting operations
- Recall that an array name without brackets is a pointer to the array's first element
- You can access the first array element using the indirection operator
- If `array[]` is a declared array,
  - the expression `*array` is the array's first element, `*(array + 1)` is the array's second element, and so on

Relationship between an Array and a Pointer (2/3)

- If you generalize for the entire array, the following relationships hold true:
  - `*(array) == array[0]`
  - `*(array + 1) == array[1]`
  - `*(array + 2) == array[2]`
  - `...`
  - `*(array + n) == array[n]`
- If `p` is pointer with
  - `p = array; /*p points to first element of array*/`

Relationship between an Array and a Pointer (3/3)

- Then following relationships holds
  - `*p == array[0]`
  - `*(p + 1) == array[1]`
  - `*(p + 2) == array[2]`
  - `...`
  - `*(p + n) == array[n]`
  - `p == &array[0]`
  - `p + 1 == &array[1]`
- `p[n]` is same as `*(p+n),` and this is same as `array[n]`
- `&p[n]` is same as `(p+n),` and this is same as `&array[n]`
- Hence pointer notation and array notation can be used interchangeably

Example

```
int x[10] = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};
int i;

for (i = 0; i <= 9; ++i)
{
   /* display an array element */
   printf("i=%d   x[i]= %d    *(x+i)= %d",
           i, x[i], *(x+i));

   /* display the corresponding array address */
   printf("  &x[i]=%p   x+i=%p\n", &x[i], (x+i));
}
```

Uses both array and pointer notation to access array elements and their addresses

Extra Example

Both function below print the contents of the array x

```
void printarray(int x[], int n)
{
  int i;
  for (i = 0; i < n; i++)
    printf("%d ", x[i]);
}
```

Array Notation

```
void printarray(int *x, int n)
{
  int i;
  for (i = 0; i < n; i++)
    printf("%d ", *(x+i));
}
```

Pointer Notation

Difference between Array and Pointer

- A pointer is a variable and it can change its value, specifically it can point to different variables at different time
- An array is a constant pointer, the address (*which is assigned by the compiler*) it points to cannot be changed
- Given the declaration

  -
    ```
    int x[10];
    int *p;
    ```
  - Constructions such as `p = x` and `p++` is legal but `x = p` or `x++` are illegal

Functions Receiving Arrays

- Recall when an array name is passed to a function, the address of the first element is passed
- Within the function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address

```
int strlen(char *s)
{
  int n;
  for (n = 0; *s != '\0'; s++)
     n++;
    return n;
}
```

Since s is a pointer, s++ has no effect on the character string in the function that called strlen, but merely increments strlen's private copy of the pointer

More Example

This strlen version uses pointers to find the length of the string

```
int strlen(char *s)
{
  char *p = s;
  while (*p != '\0')
     p++;
  return p-s;
}
```

In its declaration, p is initialized to s, that is, to point to the first character of the string. In the while loop, each character in turn is examined until the '\0' at the end is seen. Because p points to characters, p++ advances p to the next character each time, and p-s gives the number of characters advanced over, that is, the string length.

Passing part of an array

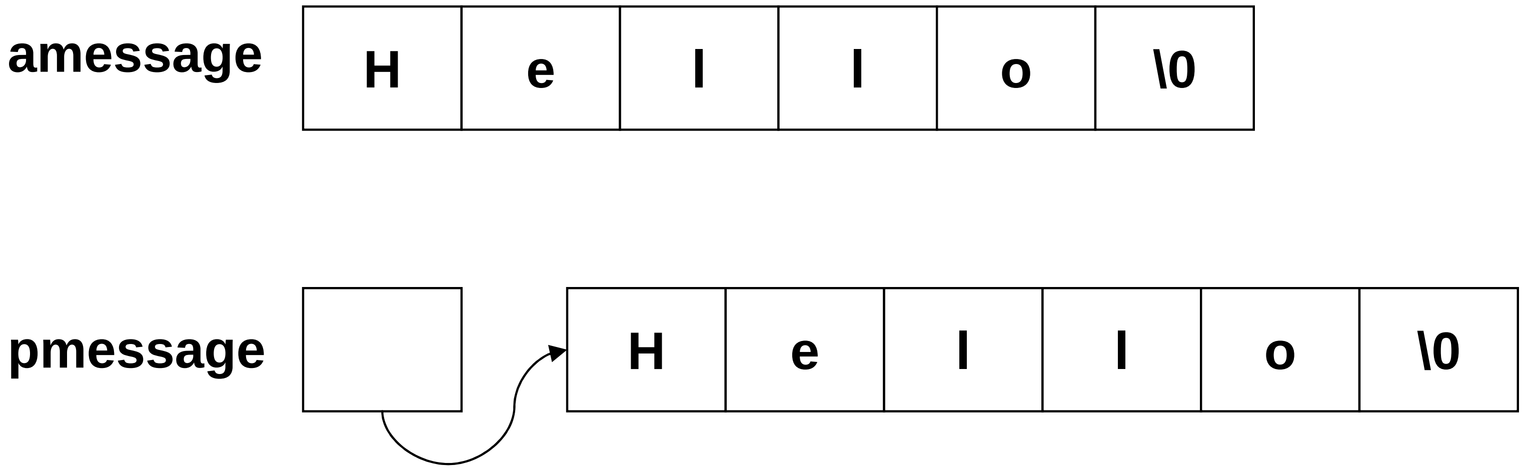
- It is possible to pass part of an array to a function, by passing a pointer to the beginning of the subarray
- For example, if `a` is an array,
  - `func(&a[2])` and `func(a+2)`
  - both pass to the function `func` the address of the subarray that starts at `a[2]`
- Within `func`, the parameter declaration can read
  - `func(int arr[]) { ... }` or `func(int *arr) { ... }`
- So as far as `func` is concerned, the fact that the parameter refers to part of a larger array is of no consequence

Character Pointers and Strings

- Recall a string constant, written as `"I am a string"` is an array of characters
- There is an important difference between these definitions:
  - `char amessage[] = "now is the time";` **/* an array */**
  - `char *pmessage = "now is the time";` **/* a pointer */**
- `amessage` is an array, just big enough to hold the string
- Individual characters within `amessage` may be changed but `amessage` will always refer to the same storage
- `pmessage` is a pointer, initialized to point to a string constant
- `pmessage` may subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents

Illustration

```
char amessage[] = "Hello";
Char *pmessage = "Hello";
```



Array of Pointers

- Since pointers are variables themselves, they can be stored in arrays just as other variables can
- The declaration for an `int` pointer array of size 10 is
  - `int *x[10];`
    This statement creates 10 int pointers
- To assign the address of an integer variable called `var` to the third element of the pointer array, write
  - `x[2] = &var;`
- To find the value of `var`, write
  - `*x[2]`

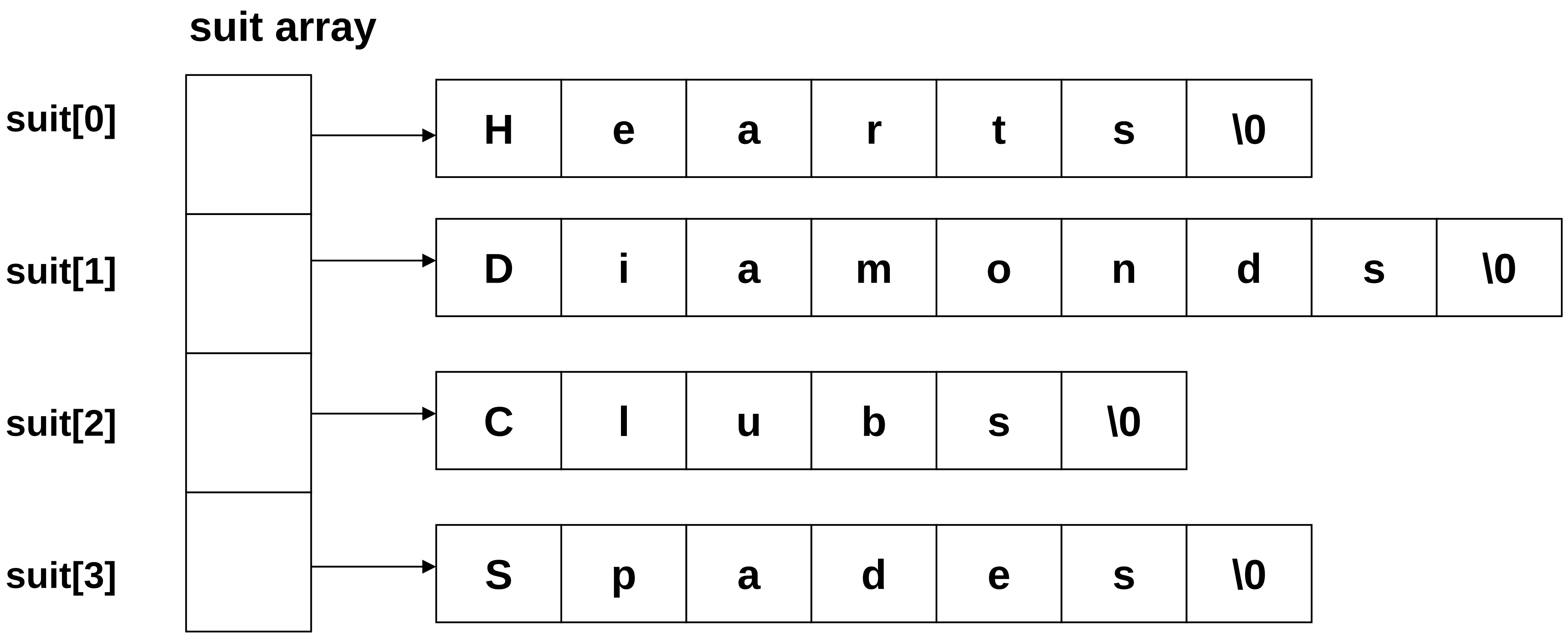
Array of Strings

- Pointer arrays are often used to hold pointers to strings
- Example

  -
    ```
    char *suit[ 4 ] = {
       "Hearts",
       "Diamonds",
       "Clubs",
       "Spades“
      };
    ```
  - Strings are pointers to the first character
  - `char *` – each element of `suit` is a pointer to a `char`

Illustration

```
char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };
```



The strings are not actually stored in the array suit, only pointers to the strings are stored

Pointer to An Array

- Skipped, See Gottfried10.7

Multiple Indirection

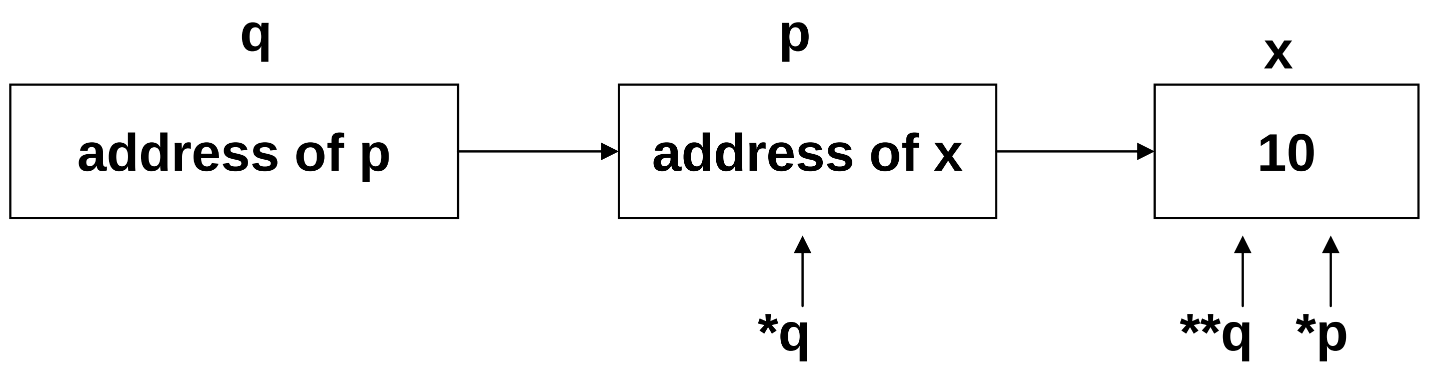
- You can create a pointer variable that stores the address of another pointer variable
- This situation is called *multiple indirection*, or *pointers to pointers*
- The following declaration tells the compiler that `q` is a pointer of type `int *`
  - `int **q;`
- To access the target value indirectly pointed to by a pointer to a pointer, you must apply the indirection operator twice

Example

```
int x, *p, **q;

x = 10;
p = &x;
q = &p;

printf("%d", **q); /* print the value of x */
```



Multiple Indirection

Dynamic Memory Allocation (1/2)

- Memory required by a program can be allocated by declaring variables in program source code
- This is called *static memory allocation*
- This requires you to know when you’re writing the program exactly how much memory you need
- *Dynamic memory allocation* is the means by which a program can allocate memory at program runtime
- DMA allows the program to react, while it’s executing, to demands for memory, such as user input

Dynamic Memory Allocation (2/2)

- To allocate memory as required, C provides library functions
- Dynamic memory is obtained from the heap
  - The heap is free memory region that is not used by your program, the operating system, or any other running program
- Global and static variables use the space of the program code
- Local variables are allocated on the stack

The malloc and free function

- The core of C’s allocation system consist of the function `malloc()` and `free()`
- These functions work together using the free memory region to establish and maintain a list of available storage
- The `malloc()` function allocates memory, and the `free()` function releases it
  - Each time `malloc()` memory request is made, a portion of the remaining free memory is allocated
  - Each time a `free()` memory release call is made, memory is returned to the system

The `malloc` function

- The `malloc()` function has this prototype:
  - `void *malloc(size_t n);`
- Here,
  - `n` is the number of bytes of memory you want to allocate. (The type `size_t` is some type of unsigned integer).
- `malloc()` returns a pointer of type `void *`, which means that you can assign it to any type of pointer
- After a successful call, `malloc()` returns a pointer to the first byte of the region of memory allocated from the heap
- If there is not enough available memory to satisfy the `malloc()` request, an allocation failure occurs and `malloc()` returns a `NULL`

Using `malloc`

- The code fragment shown here allocates 1,000 bytes of contiguous memory:

  -
    ```
    char *p;
    p = malloc(1000); /* get 1000 bytes */
    ```
- After the assignment, `p` points to the first of 1,000 bytes of free memory
- The next example allocates space for 50 integers.

  -
    ```
    int *p;
    p = malloc(50*sizeof(int));
    ```
- Once you get the memory, you can store data in the memory using the pointer returned by `malloc`

The `free` function

- The `free()` function is the opposite of `malloc()` in that it returns previously allocated memory to the system
- Once the memory has been freed, it may be reused by a subsequent call to `malloc()`
- The function `free()` has this prototype:
  - `void free(void *p);`
- Here, `p` is a pointer to memory that was previously allocated using `malloc()`
- It is critical that you never call `free()` with an invalid argument; this will damage the allocated system

Using `malloc` and `free`

```
int *p;

/* allocate space for an integer */
p = malloc(sizeof(int));

if (p == NULL) { /* there was no integer */
   puts("Memory Exhausted");
   exit(1); /* terminate the program */
}

*p = 100; /* store 100 in the newly allocated memory */

printf("Address returned by malloc is %p, "
        "value stored there is %d\n", p, *p);

free(p); /* release the memory allocated by malloc */
```

Dynamically Allocated Arrays

- Since
  - the `malloc()` function returns a pointer to the first byte of contiguous memory and
  - array elements are stored contiguously in memory and
  - also you can use a pointer to index array elements,
- You can create a dynamic array whose size need not be specified in advance

```c
float *plist;
int i, n;
float sum=0;

printf("How many numbers? ");
scanf("%d", &n);

plist = malloc(sizeof(float)*n);

for (i = 0; i < n; i++) {
      printf("Enter number %d: ", i+1);
      scanf("%f", plist+i);
      sum += *(plist+i);
}

printf("The sum of the numbers ");
for (i = 0; i < n; i++)
      printf("%f ", *(plist+i));
printf("is %f", sum);

free(plist);
```

Dynamically Allocated Array of Strings

- Skipped, See book Gottfried Sec. 10.8, page 309-311