UNIT 1
Review of Programming Basics

CPE 1202L: Data Structures and Algorithms

COMPUTER ENGINEERING
UNIVERSITY of SAN CARLOS

- The **C language** is one of the most popular programming languages. The standard of C was defined in 1989 with changes in 1995 and 1999. It is called ANSI/ISO Standard C. The language has:
  - Elementary and structured data types (including pointers)
  - Control constructs
  - Procedures and functions
  - Input/Output

Data Structures

- Definition
  - Data structure is a collection of data and a set of rules for organizing and accessing it.
  - The choice of data structures obviously affects the operations that can be done on the data.
  - A programmer must appropriately select the appropriate data structure given a specific problem.

FUNCTIONS

Function

- The building blocks of C in which all program activity occurs
- General form of a Function

```
type_specifier function_name(parameter_declaration_list)
{
    body of the function
}
```

- type_specifier
  - specifies the type of the value that function will return using the return statement
  - default return an integer result
- parameter_declaration_list
  - comma separated list of variable types & name that will receive the values of the arguments when function is called

Uses of the return Statement

- Causes an immediate exit from function execution and transfers control to the calling code

- It can be used to return a value

Function Arguments
No Return Value, with Parameter

```
#include<stdio.h>
void sqr (int x);
main()
{
     int num = 100;
      sqr (num);
}

void sqr (int x)
{
        prinft("%d squared is %d ", x, x*x);
       getch();
}
```

Function Arguments
No Return Value, with Parameter

```
#include<stdio.h>
void mul (int a, int b);
main()
{
    mul (10, 11);
}

void mul (int a, int b)
{
    printf(“%d”, a*b);
    getch();
}
```

Function Returning Values
With Return Value, With Parameter

```
#include<stdio.h>
int mul (int a, int b);
main()
{
  int answer;
  answer = mul (10, 11);
  printf("The answer is % d", answer);
  getch();
}

int mul (int a, int b)
{
  return a*b;
}
```

Ways of Passing Arguments CALLING FUNCTION

- **Call by Value**
  - copies value of an argument into the formal parameter of a function

- **Call by Reference**
  - the address of an argument is copied into the parameters of a function and is used to access the actual argument

Ways of Passing Arguments CALLING FUNCTION

- **Pointer Operator**
  - **Ampersand (&)**
    - unary operator that returns the memory address of its operand (“the address of ”)
  - **Asterisk (*)**
    - returns the value of variable located at the address that follows (“value at address”)

ARRAYS

Array

- Is a collection of variables of the same type placed contiguously in memory and referenced by a common name.

One-dimensional Array

- Declaration
  - data_type variable_name[array_size];
- Total number of bytes
  - size_of_data_type * size_of_array
- Example
  - int x[10]; # of bytes = 2*10 = 20
  - char y[20]; # of bytes = 1*20 = 20

Array Indexing

- Indexes
  - Used to differentiate the elements in an array
- Example
  - int num[5];

Index	Element Name	Address
0	num[0]	&num [0], &num, num
1	num[1]	&num [1]
2	num[2]	&num [2]
3	num[3]	&num [3]
4	num[4]	&num [4]

Two-dimensional Array

- In essence, list of one-dimensional arrays
- Declaration
  - data_type variable_name[row_size][col_size];
- Total number of bytes
  - size_data_type * row_size * col_size
- Example
  - int x[5][3]; # of bytes = 2*5*3 = 30
  - char y[2][5]; # of bytes = 1*2*5 = 10

Array Indexing

- Access elements
  - int x[3][2];

Index	Element Name	Address
0,0	x[0][0]	&x[0][0], &x[0], &x, x
0,1	&x[0][1]	&x[0][1]
1,0	&x[1][0], &x[1]	&x[1][0], &x[1]
1,1	&x[1][1]	&x[1][1]
2,0	&x[2][0]	&x[2][0], &x[2]
2,1	&x[2][1]	&x[2][1]

STRUCTURES

Structure

- Is a collection of variables that are referenced under one name, providing a convenient means of keeping related information together.

- Structure elements
  - Variables that make up the structures

- Uses the keyword `struct` which tells the compiler that a structure is being declared

General form of a structure declaration:

```
struct structure_tag_name{
  type variable_name;
  type variable_name;
  type variable_name;
  .
  .
  .
} structure_variables;
```

Structure declaration example

- 1)

```
struct stud
{
 char id[10];
 char name[30];
 char course[10];
 int year;
};
struct stud stud_info;
```

Structure declaration example

- 2)

```
struct stud
{
 char id[10];
 char name[30];
 char course[10];
 int year;
} stud_info, sdata;
```

Structure declaration example

- 3)

```
struct
{
   char id[10];
   char name[30];
   char course[10];
   int year;
} stud_info;
```

Referencing Structure Elements

- Individual structure elements are referenced by using the dot (.) operator
- General form:
  - `structure_name.element_name`

Example

```
gets(stud_info.id);
gets(stud_info.name);
gets(stud_info.course);
scanf("%d",&stud_info.year);
```

```
printf("%s",stud_info.id);
printf("%s",stud_info.name);
printf("%s",stud_info.course);
printf("%d",stud_info.year);
```

Array of Structures

```
#define MAX 20
struct stud
{
  char id[10];
  char name[30];
  char course[10];
  int year;
};
struct stud stud_info[MAX];
```

Array of Structures

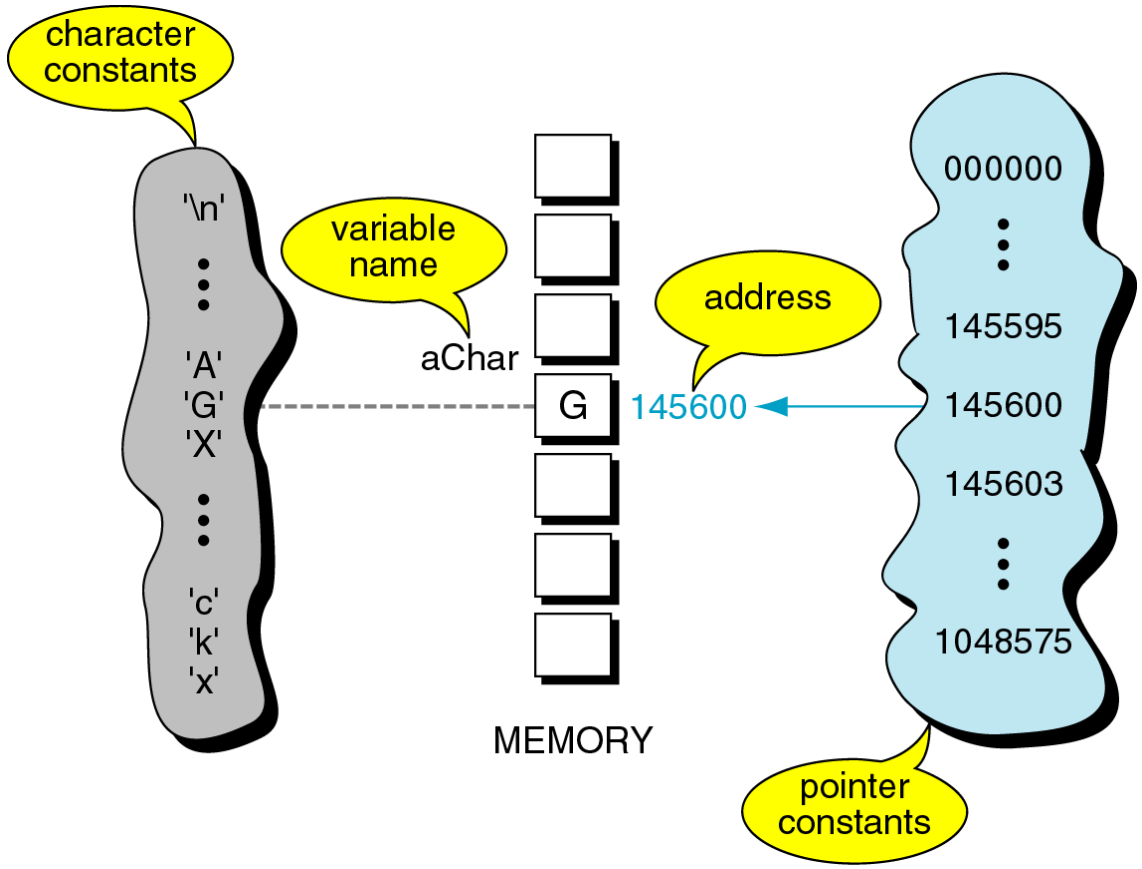
```
gets(stud_info[2].id);
gets(stud_info[2].name);
gets(stud_info[2].course);
scanf(“%d”,&stud_info[2].year);

printf(“%s”,stud_info[5].id);
printf(“%s”,stud_info[5].name);
printf(“%s”,stud_info[5].course);
printf(“%d”,stud_info[5].year);
```

POINTERS

Pointers

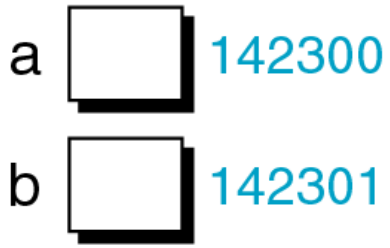
- **Pointer** is a constant or variable that contains an address that can be used to access data.

Pointer Constants

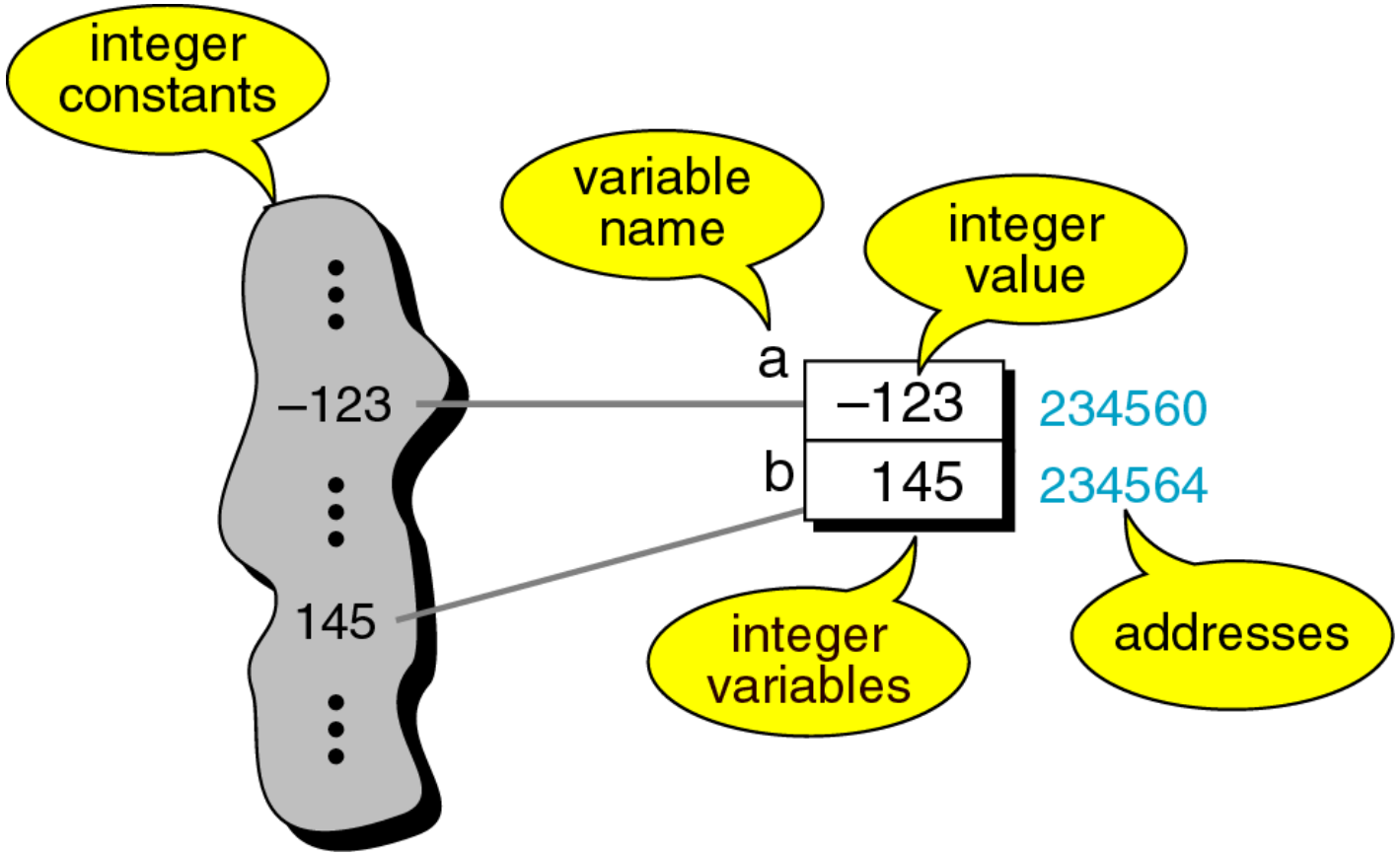


Print Character Address

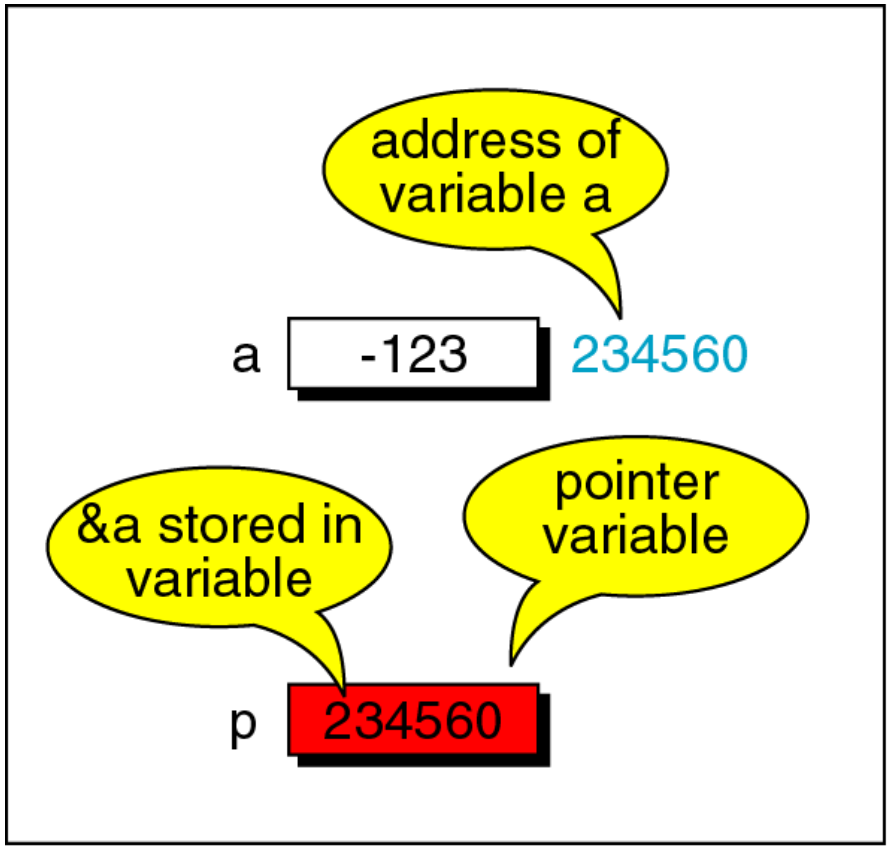
```c
/* Print character addresses */
#include <stdio.h>

int  main  (void)
{
/* Local Definitions */
   char   a;
   char   b;
/* Statements */
   printf ("%p %p\n", &a, &b);

   return 0;
}  /* main */
```



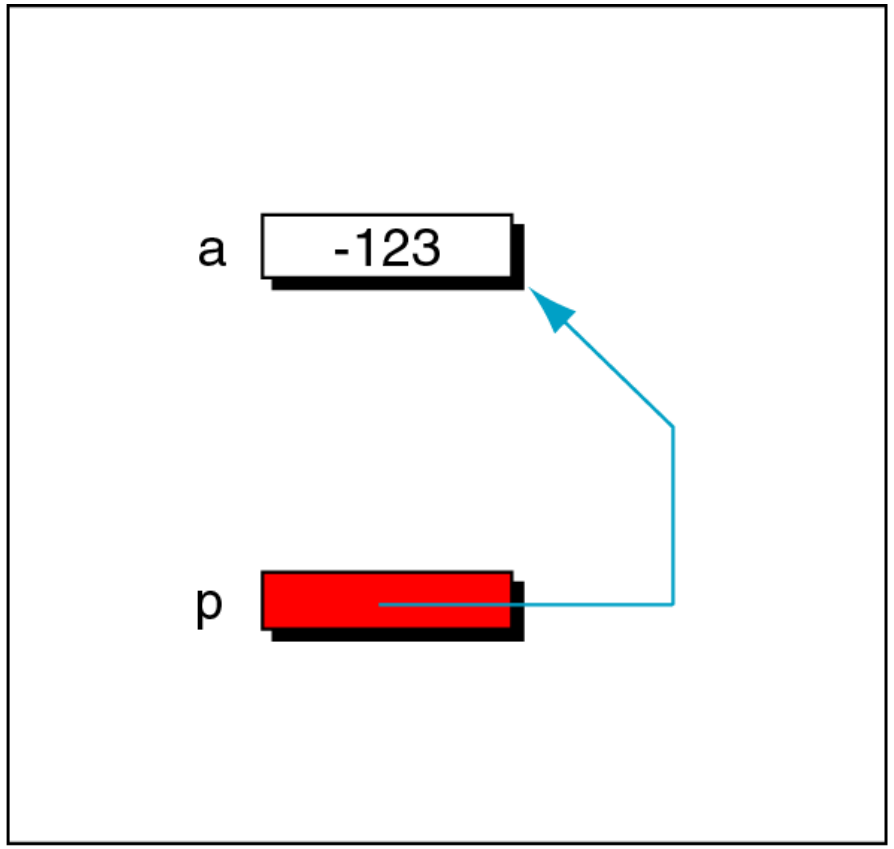
Integer Constants and Variables



Pointer Variable



Physical representation



Logical representation

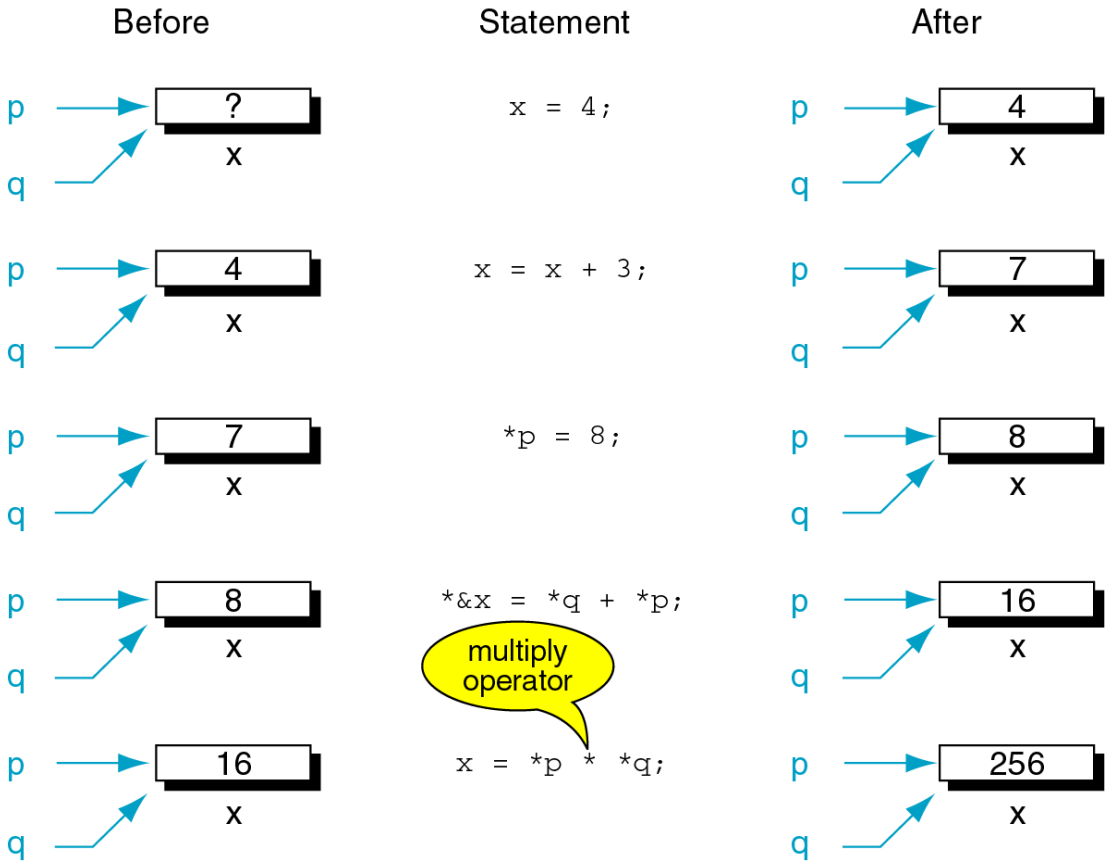
Accessing Variables through Pointers

- The indirection operator is a unary operator whose operand must be a pointer value
- An indirection expression, one of the expression types in the unary expression category is coded with an asterisk (*) and an identifier
- To access the variable a through the pointer p
  - * p
  - p = &a
  - a++; a = a + 1; *p = *p+1; (*p)++

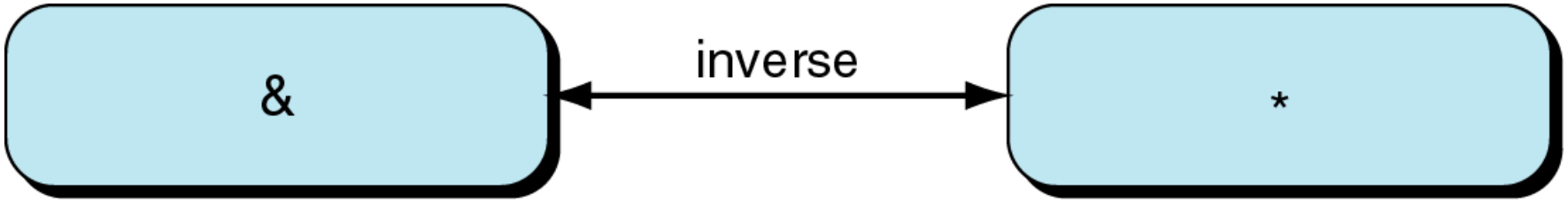
Operations on Pointers

`&i`	address of a referenced location
`*p`	contents of a referenced location
`p++`	value of the pointer p incremented by one unit of base type size (`p= p + sizeof(base type)`)
`++p`	increments the pointer `p = p + sizeof(base type)` then gets the value of `p`
`p = &i`	assigns a memory location to the pointer
`type *p`	declares a pointer and its base type
`*(p++) or *p++`	increments p and than gets a value of `*p`
`(*p)++`	is not equivalent to `*p++`
`*(++p) or *++p`	preincrements `p`, then gets the value
`++(*p)`	increments the value pointed to by `p`

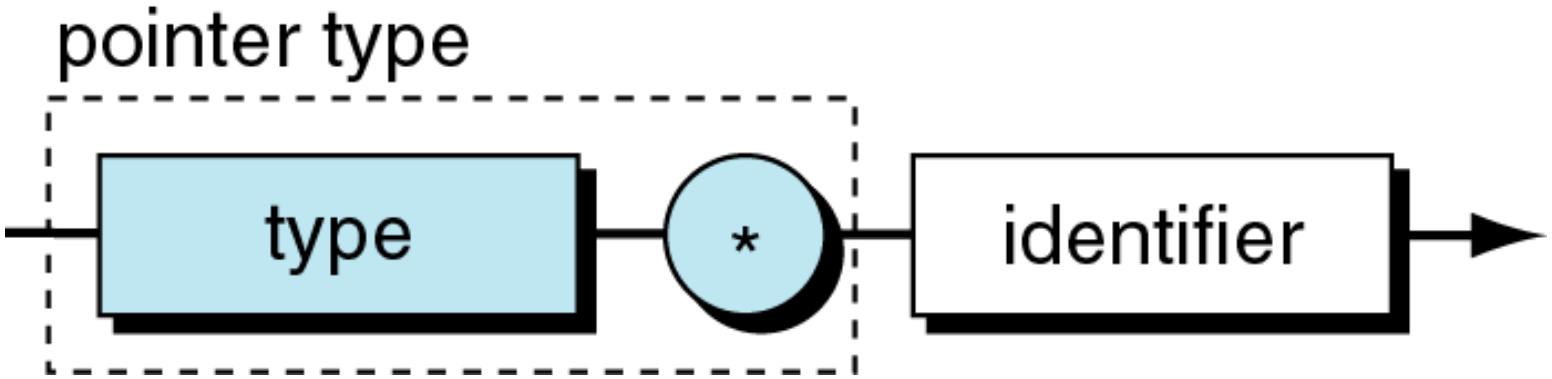
Accessing Variables through Pointers



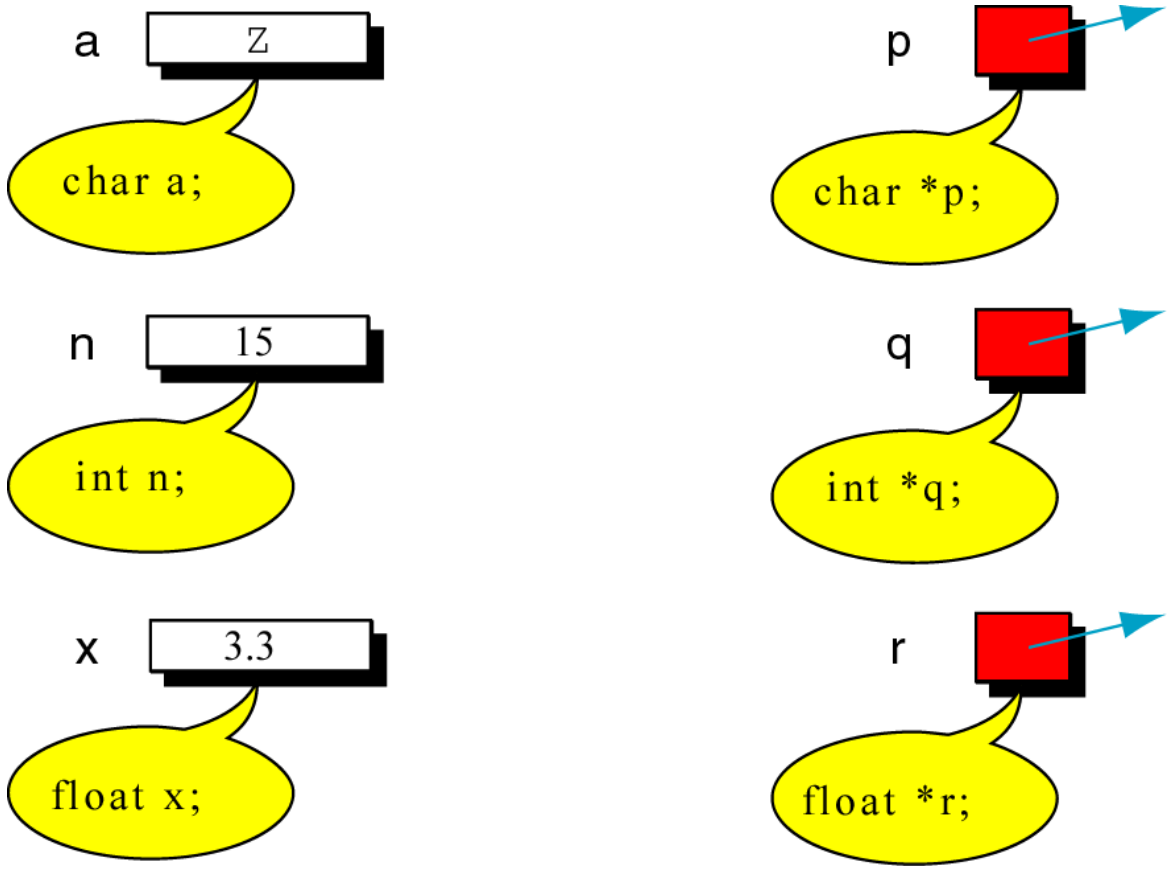
Address and Indirection Operators



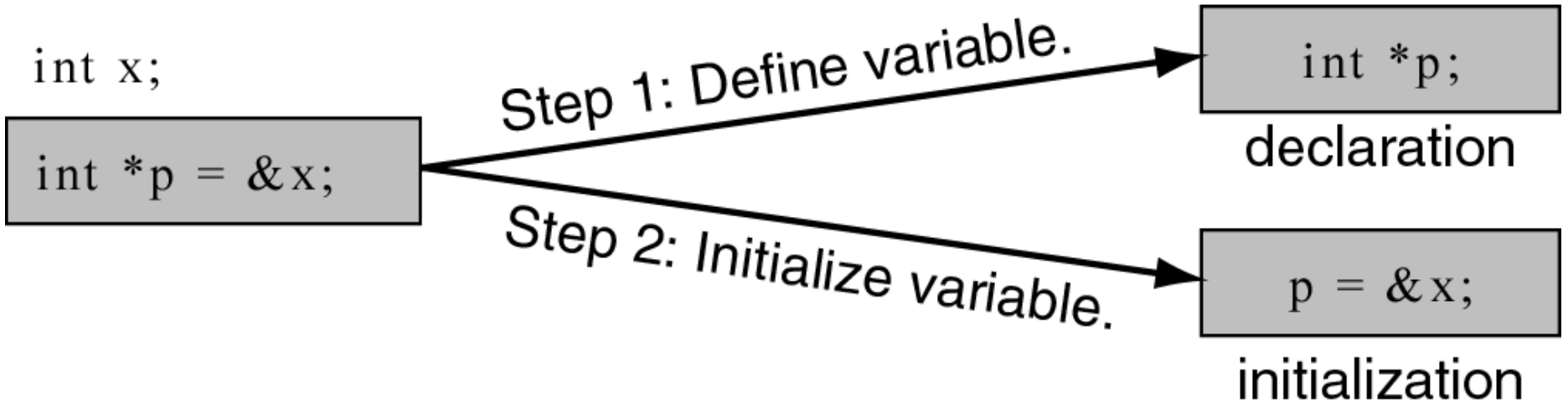
Pointer Variable Declaration



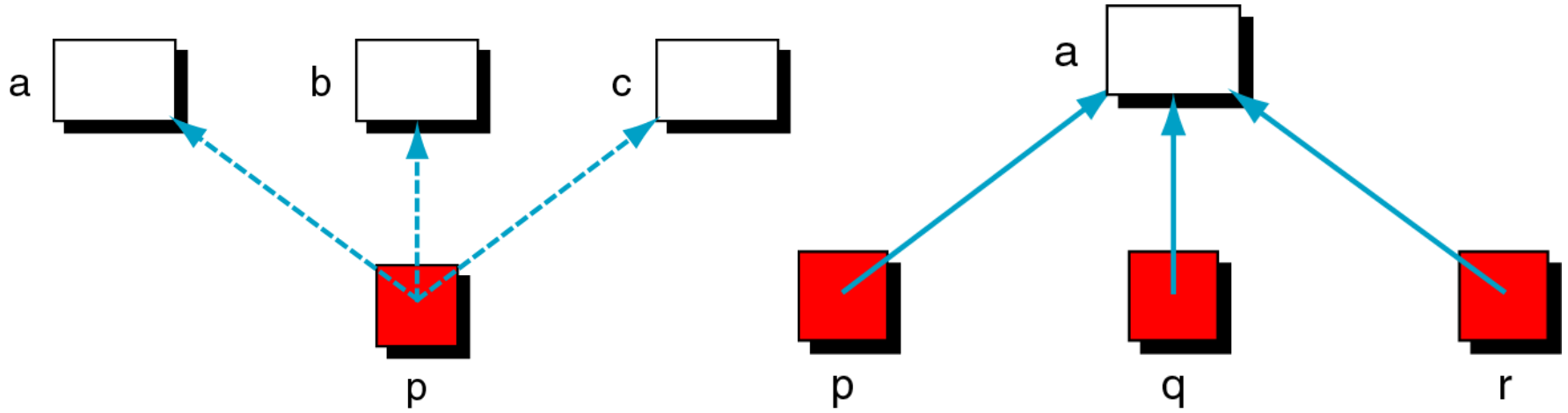
Declaring Pointer Variables



Initializing Pointer Variables



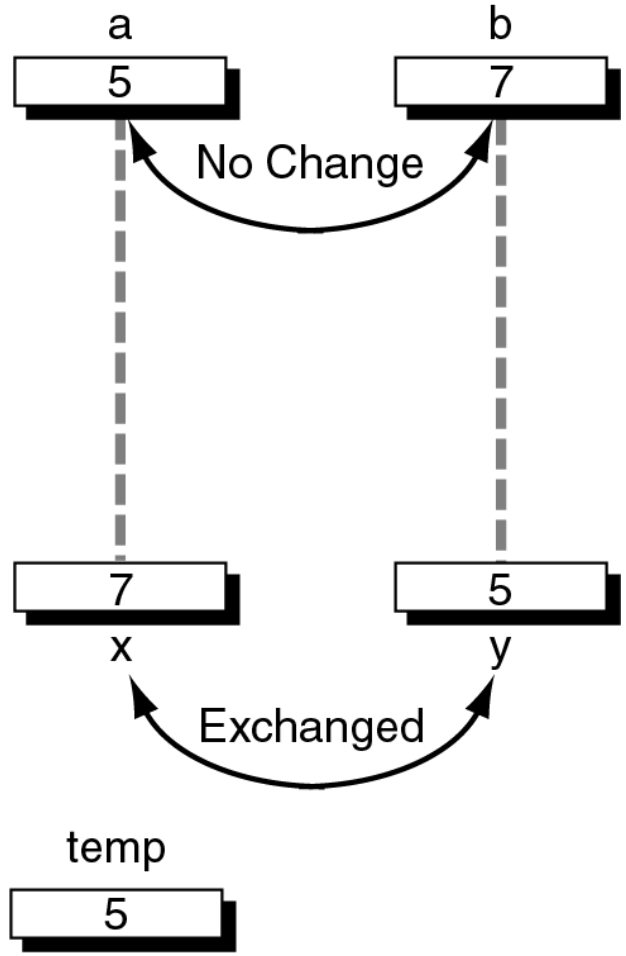
Pointer Flexibility



Pointers for Inter-Function Communication

```
/* Prototype Declarations */
void exchange (int x, int y);

int main (void)
{
  int  a = 5;
  int  b = 7;
  exchange (a, b);
  printf("%d %d\n", a, b);
  return 0;
} /* main */
```

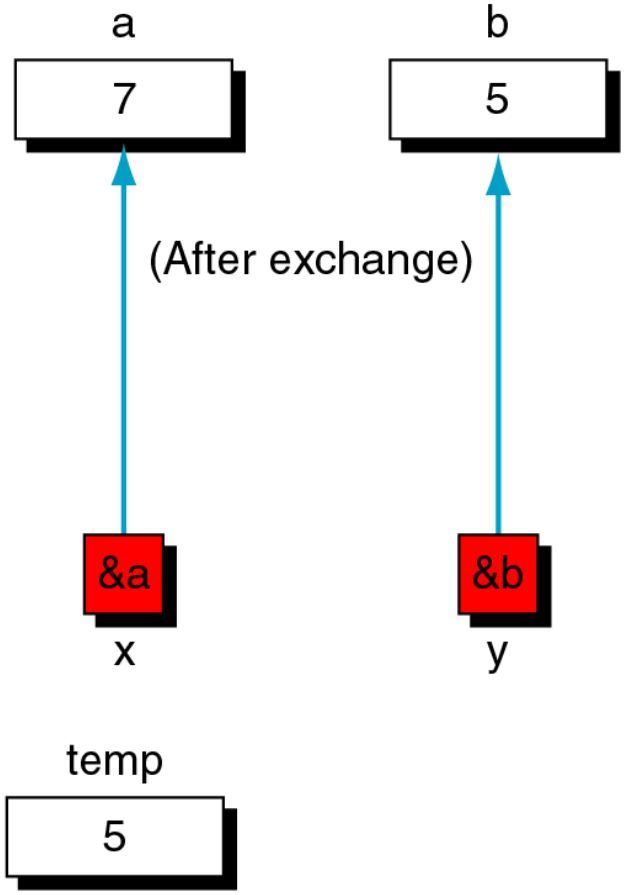
```
void exchange (int x,
               int y)
{
  int temp;

  temp = x;
  x    = y;
  y    = temp;
  return;
} /* exchange */
```



Pointers for Inter-Function Communication

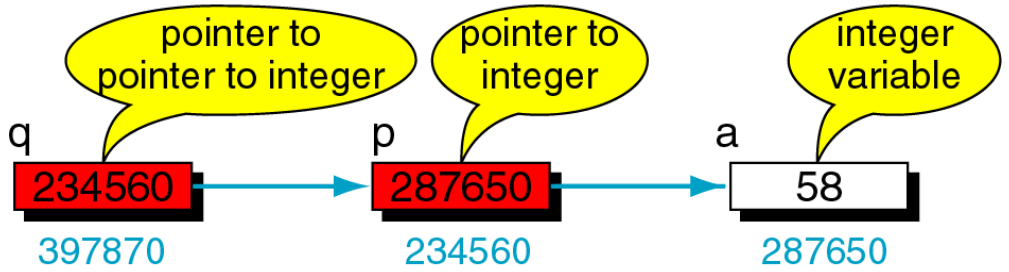
```
/* Prototype Declarations */
void exchange (int *, int *);

int main (void)
{
  int  a = 5;
  int  b = 7;
  exchange (&a, &b);
  printf("%d %d\n", a, b);
  return 0;
} /* main */
```

```
void exchange (int *x,
               int *y)
{
  int temp;

  temp = *x;
  *x   = *y;
  *y   = temp;
  return;
} /* exchange */
```



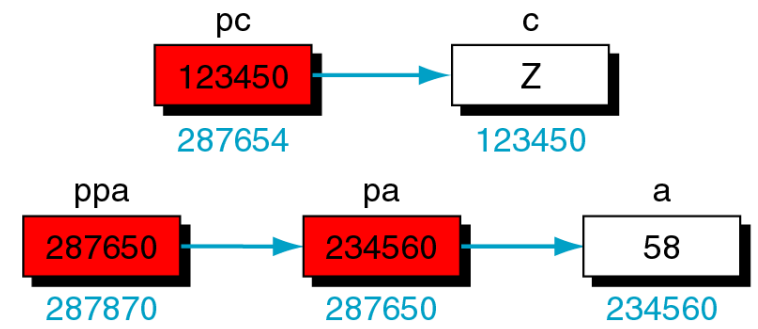
Pointers to Pointers

```
/* Local Declarations */
int       a;
int      *p;
int     **q;
```




```
/* Statements */
a = 58;
p = &a;
q = &p;
printf(" %3d",   a);
printf(" %3d",  *p);
printf(" %3d", **q);
```

Deference Type Compatibility




```
char c;
char *pc;

int   a;
int  *pa;
int **ppa;

pc  = &c;          /* Good and valid   */
pa  = &a;          /* Good and valid   */
ppa = &pa;         /* Good and valid   */

/* Invalid pointers will generate errors   */
pc  = &a;          /* Different types  */
ppa = &a;          /* Different levels */
```

Memory Allocation

- To allocate memory dynamically the function malloc is used. It returns a generic pointer of type void*. In order to obtain a correct type we must change the generic pointer type to a type we need using a cast (for instance, (int*)). The obtained pointer indicates the beginning of the allocated space. If malloc fails to allocate memory it returns NULL.

```
int *a;
a = (int *) malloc(sizeof(int));
if (a == NULL) { printf("malloc failed\n"); }
*a = 120;
```

- Usually, we do not allocate space for variables of primitive type at run time. Typically, malloc is used to allocate space for a structure or an array.

Memory Allocation

- Examples

```
typedef struct {
   int ID;
   double GPA; /* grade */
} Student;
Student *sptr;
sptr = (Student *) malloc(sizeof(Student));
sptr->ID = 2310;
sptr->GPA = 3.4;
```

Memory Allocation

- To allocate an array dynamically, we multiply sizeof(type) by the required number of elements of the array.

```
int i;
Student *p; /* p is an array */
p = (Student *) malloc(35*sizeof(Student));
for (i = 0; i < 35; i++) {
   p[i].ID = 120 + i;
   p[i].GPA = 0.0;
}
```

Memory Allocation

```
#include <stdio.h>
#include<stdlib.h>

struct name {
  int a;
  float b;
  char c[30];
};
```

Memory Allocation

```
int main()
{
   struct name *ptr;
   int i,n;

   printf("Enter n: "); scanf("%d",&n);
   ptr=(struct name*)malloc(n*sizeof(struct name)); /* Above statement allocates the memory for n
                                                       structures with pointer ptr pointing to base address */
```

Memory Allocation

```
  for(i=0;i<n;++i)
   {
         printf("Enter string, integer and floating number respectively:\n");
         scanf("%s%d%f",&(ptr+i)->c,&(ptr+i)->a,&(ptr+i)->b);
   }

   printf("Displaying Infromation:\n");
   for(i=0;i<n;++i)
         printf("%s\t%d\t%.2f\n",(ptr+i)->c,(ptr+i)->a,(ptr+i)->b);
   return 0;
}
```

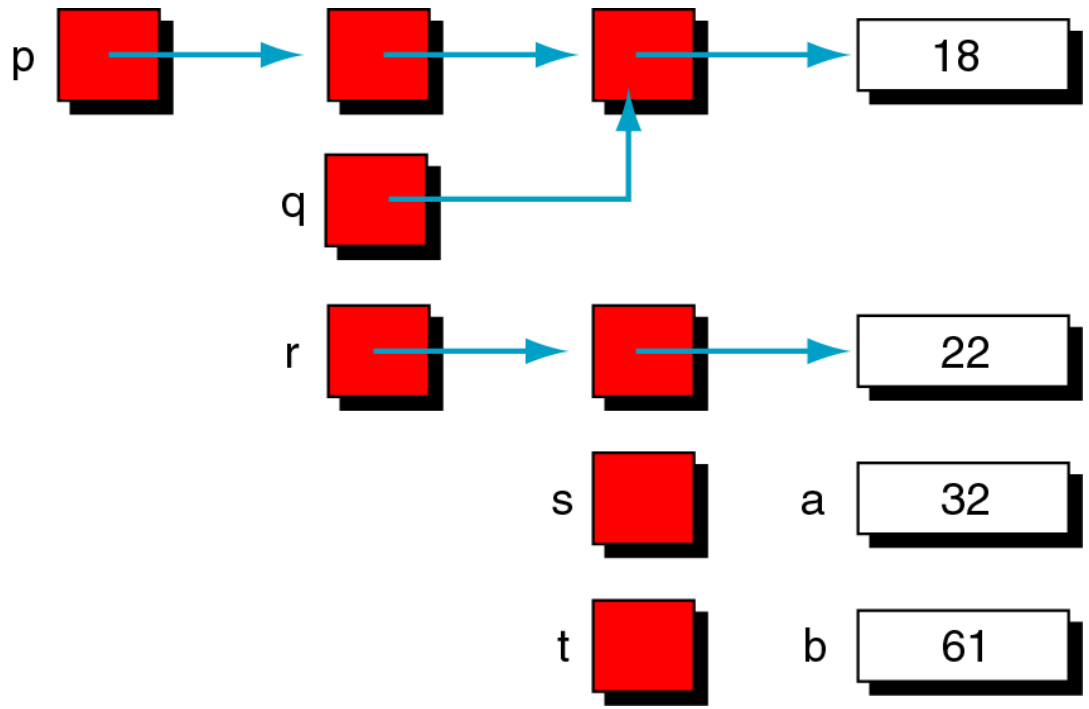
Memory Deallocation

- The allocated memory remains in use until the program has finished or memory is released using the function **`free`**.

Homework 1

- Assuming all variables are integers, and all pointers are typed appropriately, show the final values of the variables in Figure below after the following assignments:

```
a = ***p;
s = **p;
t = *p;
b = **r;
**q = b;
```



Homework 2

- Create a program that adds, subtracts, multiples, and divides two numbers using pointers.

References

- Richard F. Gilberg, Behrouz A. Forouzan, **Computer Science: A Structured Programming Approach Using C 3rd edition**, Cengage Learning Course Technology © 2006

COMPUTER ENGINEERING

UNIVERSITY *of* SAN CARLOS