Pointers

Pointers - Introduction

- What are these?
  - `int a, b, c;`
- What are these?
  - `int *p, *q, *r;`

Pointers - Introduction

- What are these?
  - `int a, b, c;`
  - `a, b, c` are names of integer variables
- What are these?
  - `int *p, *q, *r;`
  - `p, q, r` are addresses storing integer variables

Pointers - Introduction

- What are these?
  - `int a, b, c;`
  - `a = 5;` Variable a is initialised to value 5
  - &a= `0x1000;` Address storing variable named a is 0x1000
- What are these?
  - `int *p, *q, *r;`
  - `*p = 5;` Content of address p is 5
  - Address p can have value 0x2000

- Given `int *p;`
  - Is &p ok to use?
- Given `int a;`
  - Is *a ok to use?

- Given `int *p;`
  - Is &p ok to use?
  - No. This means address of address.
- Given `int a;`
  - Is *a ok to use?
  - No. This means value of value.

```
char my_name[50];
int a[100];

printf("\nEnter name and age: ");
scanf("%s %d", my_name, &a[10]);
```

No & for array, but & required for element of array.

Dereferencing Pointers

- Dereferencing is an operation performed to access and manipulate data contained in the memory location pointed to by the pointer
- * is the dereference operator

```
int *p1, var1;
var1 = 10;
printf(“value of var1 is %d”, var1); // 10
p1 = &var1;
*p1 += 5;
printf(“value of var1 is %d”, var1); // 15
```

Print value and address

```
int u = 5;
int v;
int *pu, *pv;
pu = &u;
v = *pu;
pv = &v;
printf("u = %d,
    &u = %p\n", u, &u);
printf("v = %d,
    &v = %p\n", v, &v);
printf("pu = %p,
    pv = %p\n", pu, pv);
```

Sample output

```
u = 5,
&u = 0xbfbef63c
v = 5,
&v = 0xbfbef638
pu = 0xbfbef63c,
pv = 0xbfbef638
```

void Pointers

- Pointers defined of type1 variables cannot hold address of type2 variables
- `float *fp; int a; fp = &a; // not allowed`
- Therefore use general purpose pointer type called the void pointer.
- `void` pointers do not have a type associated with them and hence can hold address of any datatype
- However, they cannot be directly dereferenced by using the indirection operator '*'
- We need to suitably typecast them to the required datatype: `*((int*) vptr)`

```
int ip1;
float fp1;
void *vptr;

ip1 = 10;
fp1 = 20.5;

vptr = &ip1;
printf("vptr points to %d\n", *((int *) vptr));

vptr = &fp1;
printf("vptr points to %f\n", *((float *)
vptr));
```

```
int ip1;
float fp1;
void *vptr;

ip1 = 10;
fp1 = 20.5;

vptr = &ip1;
printf("vptr points to %d\n", *((int *) vptr));

vptr = &fp1;
printf("vptr points to %f\n", *((float *) vptr));
```

```
Vptr points to 10
Vptr points to 20.500000
```

Pointer Arithmetic – Rule 1

- A pointer variable can be assigned the address of an ordinary variable

```
int v, *pv;
char c;
char str[3]="CSE";
unsigned *pun;
pv = &v;
pv = &c; /* warning:
            incompatible types */
```

Pointer Arithmetic – Rule 2

- A pointer variable can be assigned the value of another pointer variable, provided they are of same type

```
int *pin1, *pin2;
float *pfl;
pin2 = pin1;
pin1 = pfl; // gives warning
```

Pointer Arithmetic – Rule 3

- A pointer variable can be assigned a null value
- NULL is a symbolic constant

```
int *pw;
pw = NULL;
```

Pointer Arithmetic – Rule 4

- An integer quantity can be added to or subtracted from a pointer variable
- The amount of increment depends on the type of the pointer variable

```
int *pw;
pw++;
pw = pw + 3;
```

Pointer Arithmetic – Rule 4

- An integer quantity can be added to or subtracted from a pointer variable
- The amount of increment depends on the type of the pointer variable

```
int *pw;
pw++;
pw = pw + 3;
```

```
Address of pw           : 0xe29ac0
After pw++, address is: 0xe29ac4
After pw+3, address is: 0xe29ad0
We get 4x3=12 locations incremented
```

Pointer Arithmetic – Rule 5

- One pointer variable can be subtracted from another pointer variable
- This makes sense only if both pointer variables point to elements in the same array

```
int *p1, *p2, a[100];
p1 = &a[51];
p2 = &a[55];
printf("\nAddress of p1: %p\n",p1);
printf("\nAddress of p2: %p\n",p2);
printf("\np1 - p2 is: %d\n",p1 - p2);
printf("\np2 - p1 is: %d\n",p2 - p1);
```

Pointer Arithmetic – Rule 5

```
int *p1, *p2, a[100];
p1 = &a[51];
p2 = &a[55];
...
```

If two arrays are different, p1=&a[51], p2=&b[55], the answer will not make any sense.

```
Address of p1: 0xbf9827b4
Address of p2: 0xbf9827c4
p1 - p2 is: -4
p2 - p1 is: 4
```

Pointer Arithmetic – Rule 6

- Two pointer variables can be compared if they point to objects of the same type

```
int *p1, *p2, a, b;
Char c[4];
printf("\np1 < p2 gives: %d", p1 < p2);
printf("\np1 > p2 gives: %d", p1 > p2);
printf("\np1 = p2 gives: %d", p1 == p2);
/* follo. gives warning without typecast */
printf("\np1 < c gives: %d", p1 < c);
printf("\np1 < (int*)c: %d", p1 < (int*)c);
```

Rules for - DO NOT

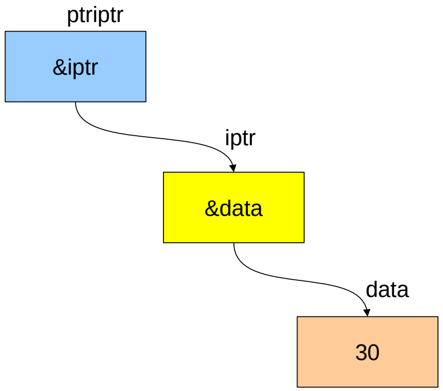
- Pointer variables cannot be multiplied by a constant
- Pointer variables cannot be added
- Ordinary variables cannot be assigned an arbitrary address

```
int x, *p1, *p2;
&x = 0xbbccddee; // error
p1 = p1 + p2;    // invalid
p1 = p2 * 2;     // invalid
p1 = p1 - p2;    // warning: p1 is int*and
                 // p1-p2 is an int
p1 = (int*)(p1-p2); // passed compilation
```

Pointers to Pointers

- We have pointer to int (datatype)
- 'pointer' is also a datatype and hence we can have pointer to a pointer

```
int *iptr;
int **ptriptr;
int data;
data = 10;
iptr = &data;
ptriptr = &iptr;
*iptr = 20;
**ptriptr = 30;
```



Function calls – without pointer

- `int my_func(int p);`
- This can be called as follows:

```
int a, b;
b = my_func(a);
     int *a; int b;
     b = my_func(*a);
int *a, *b;
*b = my_func(*a);
```

Function calls – with pointer

- `int my_func(int *p);`
- This can be called as follows:

```
int a, b;
b = my_func(&a);
     int *a; int b;
     b = my_func(a);
int *a, *b;
*b = my_func(a);
```

Function Calls – For Arrays

- Arrays are passed by reference
- Pass address of the array to the function
- Function uses this address to access each element of the array and read/modify it
- Implication: If you change the value inside the function, it gets reflected in the calling function
- Multidimensional arrays:
  - Example: read matrix, number of row and columns
  - Code on next page

Example 1 : pass array to function

```
#include <stdio.h>
void read_input (int a[][5],
                  int*, int*);
main() {
  int row, col, a[5][5];
  read_input(a, &row, &col);
}

void read_input(int a[][5],
            int *m, int *n)
{
  int i, j;
```

```
  printf("\nHow many rows? ");
  scanf("%d", m);
  printf("\nHow many cols? ");
  scanf("%d", n);

printf("\nInput the matrix
           elements:\n");
  for(i=0 ; i < *m ; i++)
    for(j=0 ; j < *n ; j++)
      scanf("%d", &a[i][j]);
}
```

Example 2 : pass array to function

```
#include <stdio.h>
void arrayArrange
(int *, int);
main() {
  int a[5], i, n;
  n = 5;
  printf("Enter the 5
    array elements:\n");
  for(i=0 ; i<5 ; i++)
    scanf("%d", a+i );
  arrayArrange(a,n);
```

```
printf("\nThe array after
       arrangement is:\n");
for(i=0 ; i<5 ; i++)
  printf("%5d", *(a+i));  }

void arrayArrange
(int *b, int k)  {
  int j;
  for(j=0 ; j<k ; j++){
    b[j] = b[j+1];
    j++;
  }}
```

Example 2 : pass array to function

```
#include <stdio.h>
void arrayArrange
(int *, int);
main() {
  int a[5], i, n;
  n = 5;
  printf("Enter the 5
    array elements:\n");
  for(i=0 ; i<5 ; i++)
    scanf("%d", a+i );
  arrayArrange(a,n);
printf("\nThe array after
       arrangement is:\n");
for(i=0 ; i<5 ; i++)
  printf("%5d", *(a+i));  }
```

```
void arrayArrange
(int *b, int k)  {
  int j;
  for(j=0 ; j<k ; j++){
    b[j] = b[j+1];
    j++;
  }}
```

```
Enter the 5 array elements:
1 2 3 4 5

The array after arrangement is:
   2   2   4   4   5
```

Function returns pointer

```
int *scan(int b[]);
int z[10];
int *a;
a = scan(z);
printf("a[0]=%d,
 a[1]=%d",a[0],a[1]);
```

```
int *scan(int
  t[])
{
   int *ret;
   t[0] = 55;
   t[1] = 66;
   ret = t;
   return(ret);
}
```

Function returns pointer

```
int *scan(int b[]);
int z[10];
int *a;
a = scan(z);
// Now a and z point to
// same array
printf("a[0]=%d,
 a[1]=%d",a[0],a[1]);
z[3] = 8;
printf("a[3]=%d", a[3]);
```

```
int *scan(int
  t[])
{
   int *ret;
   t[0] = 55;
   t[1] = 66;
   ret = t;
   return(ret);
}
```

Function returns pointer

```
int *scan(int b[]);
int z[10];
int *a;
a = scan(z);
printf("a[0]=%d,
 a[1]=%d",a[0],a[1]);
z[3] = 8;
printf("a[3]=%d",
  a[3]);
```

```
int *scan(int t[])
{
  int *ret;
  t[0] = 55;
  t[1] = 66;
  ret = t;
  return(ret);
```

```
a[0] = 55, a[1] = 66, a[3] = 8
```

Array of pointers

```
#include <stdio.h>

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

  int i, *ptr[3];

  printf("a = %p,
          &a[0] = %p",
          a, &a[0]);
```

```
for(i=0 ; i<3 ; i++) {
  ptr[i] = a + i;
  printf("\na[i] = %d,
  a+i = %p, ptr[i] = %p",
  a[i], a+i, ptr[i]);
}

for(i=0 ; i<3 ; i++)
  printf("\nptr[i] = %p,
  *ptr[i] = %d",
  ptr[i], *ptr[i]);
}
```

Array of pointers

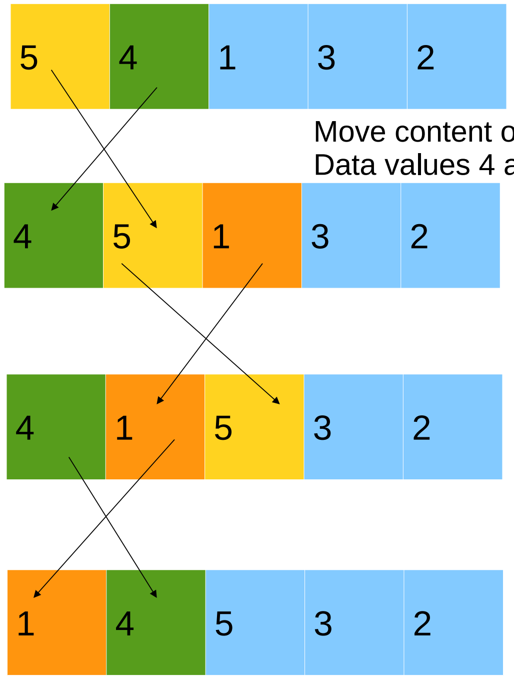
```
a = 0xbfa47750, &a[0] = 0xbfa47750

a[i] = 1, a+i = 0xbfa47750, ptr[i] = 0xbfa47750
a[i] = 2, a+i = 0xbfa47754, ptr[i] = 0xbfa47754
a[i] = 3, a+i = 0xbfa47758, ptr[i] = 0xbfa47758

ptr[i] = 0xbfa47750, *ptr[i] = 1
ptr[i] = 0xbfa47754, *ptr[i] = 2
ptr[i] = 0xbfa47758, *ptr[i] = 3
```

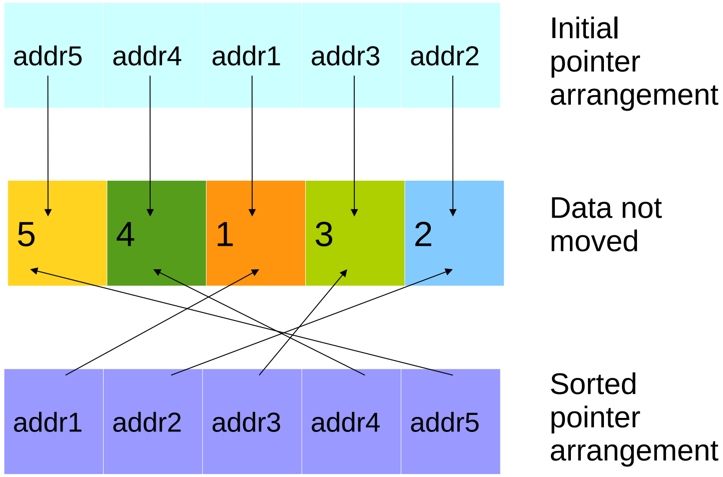
Passing array of pointers to functions

- Why? -- Consider the following scenario:
- Pass an array to function to manipulate data
- Use temporary variable to store value before swap
- Temporary storage required of the size of the datatype of the array
- Instead of swapping data, swap pointers



Move content of cells.
Data values 4 and 5 are swapped

Move pointer instead of data



Pass Array of pointers to function

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void sortbyptrexchange
(char **list, int n);
main()
{
  int  i;
  char *names[5];
  printf("\nEnter 5 names:\n");

  for(i=0 ; i<5 ; i++) {
    // allocate space to store names
    // of max length 20
    names[i] = (char *) malloc(20);
    scanf("%s", names[i]); }
  sortbyptrexchange(names, 5); ... }
```

- Define names as array of pointers to character. It will hold 5 elements. Each cell has address pointing to a character
- Allocate space to store each name using malloc. This will return an address of starting location of name. Store this in name[i]
- Pass array of pointers to function

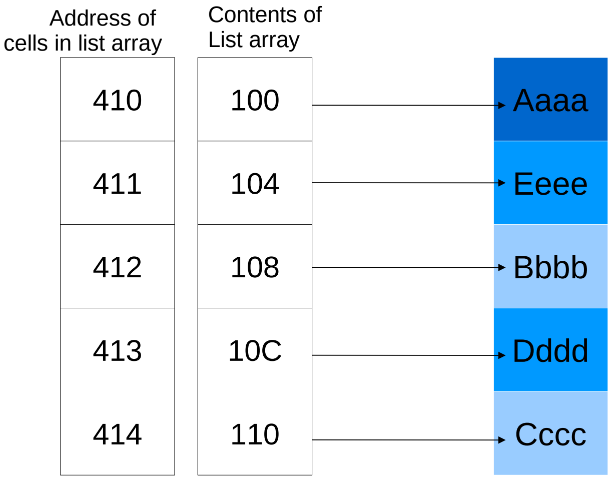
Pass Array of pointers to function

```
void sortbyptrexchange
(char **list, int n)
{
  int i, j;
  char *tempptr;

  for(i=0 ; i<n ; i++)
    for(j=0 ; j< n-1 ; j++)
      if(strcmp
        (list[j], list[j+1]) > 0)
        {
          // exchange pointers
          tempptr   = list[j];
          list[j]   = list[j+1];
          list[j+1] = tempptr;
        }
}
```

- Define temporary storage to keep the address pointing to character: tempptr
- Sort the strings and if movement required, move the addresses within the list array

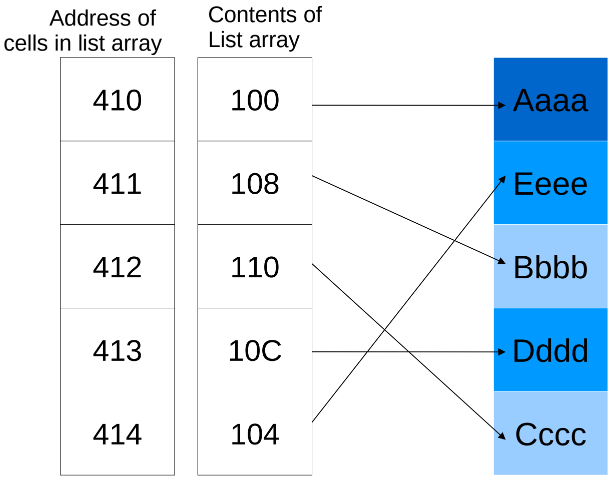
Contents of list array BEFORE sorting



Address of cells in list array	Contents of List array
410	100
411	104
412	108
413	10C
414	110

Assuming each string pointed to is 4 locations long

Content of list array AFTER sorting



Address of cells in list array	Contents of List array
410	100
411	108
412	110
413	10C
414	104

Assuming each string pointed to is 4 locations long

Pointers and 2D Array

- A two-dimensional array is a collection of CONTIGUOUS one-dimensional arrays
- int x[10][20]; can we written as

int(*x)[20];

  - x points to the first element of the zero'th row, which has 20 elements
  - (x+1) points to the first element of the one'th row
  - x[2][5] is equivalent to *(*(x+2)+5)

- a[row][col] = *(*(a+row)+col)
- scanf(“%d”, &a[row][col]);
- scanf(“%d”,(*(a+row)+col));
- printf(“%d”, a[row][col]);
- printf(“%d”,*(*(a+row)+col));

Expressions equivalent to a[i][j]

- Int a[3][5]; // declaration
- *(a[i] + j)
- (*(a+i))[j] // (a+i) points to 1st

            // element of ith row
- *((*(a+i)) + j)
- *(&a[0][0] + 5*i + j)
- Array name a is equivalent to &a[0]: pointer to an array of 5 ints
- Base address of array is &a[0][0]

Strings and Pointers

- char *name – is a pointer to character
- char *name[ ] - is an array of strings

```
char chArr[ ] = “Sample string”;
char *chPtr;
chptr = chArr; // chPtr points to chArr
```

- char name[10][12] – array of 10 strings, each 12 characters long.

Strings and Pointers

- `char *name[10] – array of pointers, name[0], ..., name[9] are pointers`
  - Different from two dimensional array
  - 2D arrays are contiguous 1D arrays
- Array of strings need not be contiguous 1D arrays
- char *colour[2] = {“red”, “black”};
  - Creates array of 2 pointers
  - Each element points to an array of characters of length 4 and 6 resp.

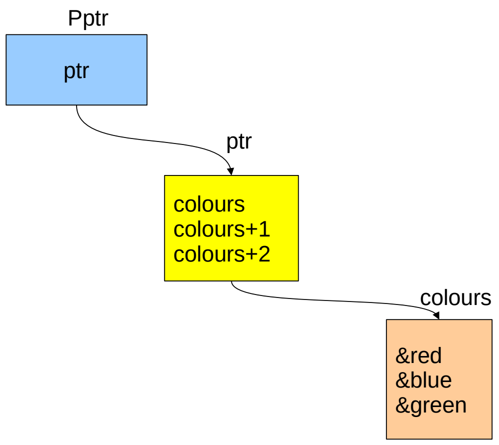
```
   // array of strings
char *colours[3] = {"red", "blue", "green"};
   // array of pointers
char **ptr[] = {colours, colours+1, colours+2};
   // Pptr is location having address of ptr
char ***Pptr = ptr;
```

*colours = red,   *(colours+1) = blue

&colours[0] = 0xbffb6240, ptr[0] = 0xbffb6240

&ptr = 0xbffb6234, ptr = 0xbffb6234, Pptr = 0xbffb6234

&Pptr = 0xbffb624c, *Pptr = 0xbffb6240, **Pptr = red



Difference between 2D array and array of pointers

- 2D arrays:
  - int x[10][20];
  - int (*x)[20];
  - char mystr[10][20];
  - char (*mystr)[20];
- Array of pointers:
  - int *x[10];
  - char *mystr[10];

- int (*data)[10];
  - data is a pointer to an array of ten elements of integer datatype
  - there is only ONE pointer
  - data++ will result in (10 * sizeof(int)) points beyond the array
- int *data[10];
  - data is an array of ten pointers of integer datatype
  - there are TEN pointers
  - data++ is invalid
  - data[0]++ will point to the next element in the 0th row, i.e. 1*sizeof(int) will be added to data[0]

Example for previous slide

```
int (*aptr)[4];
int *yptr[2];
int x[2][4] = {1, 2, 3, 4, 5, 6, 7, 8};
int i, j;

aptr = x;

printf("\n &x[0] = %p, x=%p, *aptr = %p \n", &x[0], x, *aptr);
 printf("\n &x[0][0] = %p, *aptr = %p \n", &x[0][0], *aptr);
printf("\n &x[1][0] = %p, *(aptr+1) = %p \n", &x[1][0], *(aptr+1));

 for(i=0; i<2 ; i++) {
   for(j=0 ; j<4 ; j++)
     printf("(*aptr)[j] = %d", (*aptr)[j]);
   aptr++;
   printf("\n"); }
```

```
int (*aptr)[4];   int *yptr[2]; int i, j;
int x[2][4] = {1, 2, 3, 4, 5, 6, 7, 8};

aptr = x;

printf("\n &x[0] = %p, x=%p, *aptr = %p \n", &x[0], x, *aptr);
 printf("\n &x[0][0] = %p, *aptr = %p \n", &x[0][0], *aptr);
printf("\n &x[1][0] = %p, *(aptr+1) = %p \n", &x[1][0], *(aptr+1));

 for(i=0; i<2 ; i++) {
   for(j=0 ; j<4 ; j++)
     printf("(*aptr)[j] = %d", (*aptr)[j]);
   aptr++;
   printf("\n"); }
```

```
 &x[0] = 0xbfdb8e6c, x=0xbfdb8e6c, *aptr = 0xbfdb8e6c

 &x[0][0] = 0xbfdb8e6c, *aptr = 0xbfdb8e6c

 &x[1][0] = 0xbfdb8e7c, *(aptr+1) = 0xbfdb8e7c
(*aptr)[j] = 1(*aptr)[j] = 2(*aptr)[j] = 3(*aptr)[j] = 4
(*aptr)[j] = 5(*aptr)[j] = 6(*aptr)[j] = 7(*aptr)[j] = 8
```

Example for previous slide

```
for(i=0 ; i<2 ; i++)
   yptr[i] = x[i];

printf("\nPrinting array using
  form a[i][j]\n");

for(i=0; i<2 ; i++) {
  printf("\n yptr[i] = %p,
   *yptr[i] = %d", yptr[i],
   *yptr[i]);
  for(j=0 ; j<4 ; j++) {
    printf("\n yptr[i][j] = %d",
    yptr[i][j]);
    printf(" *(yptr[i] + j) = %d",
     *(yptr[i] + j));
  } }
```

```
printf("\nPrinting array
  using form a[i]++\n");

for(i=0; i<2 ; i++) {
 for(j=0 ; j<4 ; j++) {
  printf("\nyptr[i] =
 %p,
  *yptr[i] = %d",
  yptr[i], *yptr[i]);


   yptr[i]++;
  } }
   printf("\n");
}
```

```
Printing array using form a[i][j]

 yptr[i] = 0xbfdb8e6c, *yptr[i] = 1
 yptr[i][j] = 1 *(yptr[i] + j) = 1
 yptr[i][j] = 2 *(yptr[i] + j) = 2
 yptr[i][j] = 3 *(yptr[i] + j) = 3
 yptr[i][j] = 4 *(yptr[i] + j) = 4
 yptr[i] = 0xbfdb8e7c, *yptr[i] = 5
 yptr[i][j] = 5 *(yptr[i] + j) = 5
 yptr[i][j] = 6 *(yptr[i] + j) = 6
 yptr[i][j] = 7 *(yptr[i] + j) = 7
 yptr[i][j] = 8 *(yptr[i] + j) = 8

Printing array using form a[i]++

 yptr[i] = 0xbfdb8e6c, *yptr[i] = 1
 yptr[i] = 0xbfdb8e70, *yptr[i] = 2
 yptr[i] = 0xbfdb8e74, *yptr[i] = 3
 yptr[i] = 0xbfdb8e78, *yptr[i] = 4
 yptr[i] = 0xbfdb8e7c, *yptr[i] = 5
 yptr[i] = 0xbfdb8e80, *yptr[i] = 6
 yptr[i] = 0xbfdb8e84, *yptr[i] = 7
 yptr[i] = 0xbfdb8e88, *yptr[i] = 8
```

Pointers to Functions

- As we have pointers to datatypes, we can have pointers to functions
- Name of the function is its address
- Situation: Various arithmetic functions and one summary function. We can send pointer of different arithmetic functions to the summary function in different situations
- Pointer to function assignment: complete function type must match the pointer type and argument list

Invoke function using pointers

```
#include <stdio.h>

int mult(int a, int b) {
  int ans;
  ans = a * b;
  return ans;
}

void main() {
  // pointer to function
  // with prototype of mult
  int (*funcptr) (int, int);

  int n, m, result;
```

```
// assign address of function
// mult to funcptr
funcptr = mult;

printf("\nenter two numbers:");
scanf("%d %d", &m, &n);

// call function mult
// using funcptr
result = funcptr(m, n);
// same as
// result = (*funcptr)(m,n);

printf("\nresult is: %d\n",
result); }
```

Passing function address to other functions

- Two functions: largest, smallest
- Function 'select' to print the result. Result is obtained by invoking a function that is sent as parameter to select

```
#include <stdio.h>

// function prototypes
int largest (int *c, int);
int smallest (int *c, int);

int largest(int *l, int m) {
  int big, i;
  big = l[0];

  for(i=0 ; i< m ; i++)
    if(l[i] > big)
      big = l[i];

  return big;
}
```

```
int smallest(int *s, int m)
{
  int small, i;
  small = s[0];

  for(i=0 ; i< m ; i++)
    if(s[i] < small)
      small = s[i];

  return small;
}
```

```
// last parameter of this
// function is a pointer to
// a function
void select (int *a1, int m1,
  int (*funcPtr)
      (int *a2, int m2))
{
  int ans;

  // call the function whose
  // address we got as
  // parameter
  ans = funcPtr(a1, m1);

  printf("\nResult in function
  select is: %d\n", ans);
}
```

```
void main()
{
  int i, n, a[5];
  int (*genFuncPtr)
      (int *, int);

  printf("Enter the 5
 numbers:");
  for(i=0 ; i<5 ; i++)
    scanf("%d", &a[i]);

  printf("&largest=%p,
          largest=%p",
  &largest, largest);

  printf("smallest=%p,
          smallest=%p",
  &smallest, smallest);
```

```
genFuncPtr = largest;
printf("Call the select function to print largest number\n");
printf("&genFuncPtr=%p, genFuncPtr=%p\n",
        &genFuncPtr, genFuncPtr);
select(a, 5, genFuncPtr);

genFuncPtr = smallest;
printf("Call the select function to print smallest number\n");
printf("&genFuncPtr=%p, genFuncPtr=%p\n",
        &genFuncPtr, genFuncPtr);
select(a, 5, genFuncPtr);  }
```

```
Enter the 5 numbers: 2 8 4 7 5

&largest=0x8048464, largest=0x8048464
&smallest=0x80484aa, smallest=0x80484aa

Call the select function to print largest number
&genFuncPtr=0xbf9eab44, genFuncPtr=0x8048464
Result in function select is: 8

Call the select function to print smallest number
&genFuncPtr=0xbf9eab44, genFuncPtr=0x80484aa
Result in function select is: 2
```