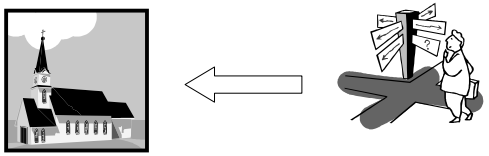
Chapter Thirteen

Pointers

1

Pointers

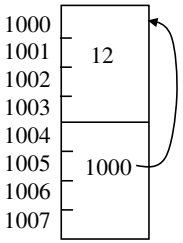
- A *pointer* is a sign used to point out the direction



2

Pointers

- A *pointer* is a data item whose value is the *address* in memory of some other value



3

Pointers

- Allow you to refer to a large data structure in a compact way
- Facilitate sharing data between different parts of a program
- Make it possible to reserve new memory during program execution
- Can be used to record relationships among data items

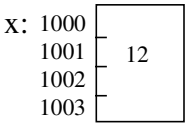
4

Variables

- Each variable refers to some location in memory and therefore has an *address*
- Once a variable has been declared, the *address* of the variable never changes, even though the *content* of the variable may change
- Depending on the *type* of data they contain, different variables require different amount of memory

5

Lvalue and Rvalue



x = x;

Store the *content* of the memory location at address 1000 to the memory location at *address* 1000

Lvalue: *address*

Rvalue: *content*

6

Lvalue-Expressions

- An *expression* that refers to a memory location capable of storing data has an *lvalue*

```
x = 1.0;
intarray[2] = 17;
```

- Many expressions do not have lvalues

```
1.0 = 1.0;        /* illegal */
x + 1.7 = 17;     /* illegal */
```

7

Lvalue-Expressions

- Each lvalue-expression refers to some location in memory and therefore has an address
- Once it has been declared, the address of an lvalue-expression never changes, even though the contents of the lvalue-expression may change
- Depending on the type of data they contain, different lvalue-expressions require different amount of memory
- *The address of an lvalue-expression is itself data that can be manipulated and stored in memory*

8

Pointer Declarations

- Pointers can be declared as
  *base-type * pointer-variable*;

```
int *iptr;
char *cptr;
int *p1, *p2;
int *p1, p2;
```

9

Pointer Operations

- & : *address-of*
  returns the *address* of an lvalue-expression

```
int  x,  *p;
p = &x;
p = &8;        /* Illegal */
```

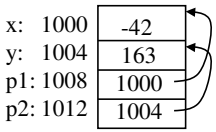
- * : *value-pointed-to* (*dereferencing*)
  refers to the memory location pointed to by a pointer

```
int  x,  *p;
p = &x;        /* *p ≡ x */
x = *p;
```

10

Examples

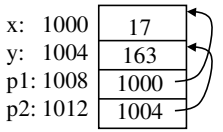
```
int x, y;
int *p1, *p2;

x = -42;
y = 163;

p1 = &x;
p2 = &y;
```



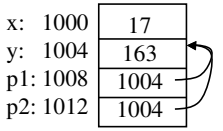
11

Examples

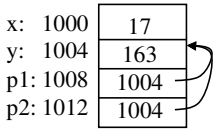
```
/* *p1 ≡ x, *p2 ≡ y */
*p1 = 17;
```




```
/* *p1 ≡ y, *p2 ≡ y */
p1 = p2;
```




```
/* *p1 ≡ y, *p2 ≡ y */
*p1 = *p2;
```



12

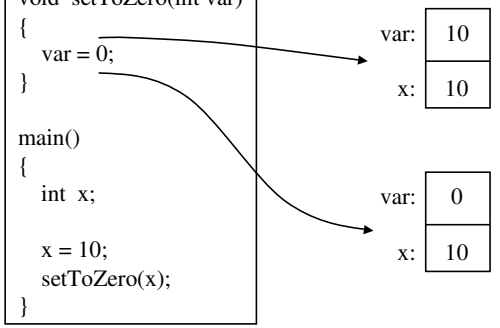
The Special Pointer NULL

- In many applications, it is useful to be able to store in a pointer variable a special value indicating that *the variable does not in fact point to any valid memory location*
- The special constant NULL is defined for this purpose
- It is important *not* to dereference a pointer variable that *has the value NULL* or *is not initialized* with the * operator

13

Passing Parameters by Value

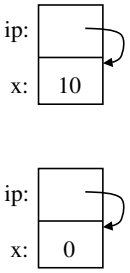
```
void  setToZero(int var)
{
   var = 0;
}

main()
{
   int  x;

   x = 10;
   setToZero(x);
}
```



14

Passing Parameters by Reference

```
void  setToZero(int *ip)
{
   *ip = 0;
}

main()
{
   int  x;

   x = 10;
   setToZero(&x);
}
```



15

An Example

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

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

16

Returning Multiple Results

```
void convertTimeToHM(int time, int *pHours, int *pMinutes)
{
   *pHours = time / MinutesPerHour;
   *pMinutes = time % MinutesPerHour;
}

main()
{
   int time, hours, minutes;
   scanf("%d", &time);
   convertTimeToHM(time, &hours, &minutes);
   printf("HH:MM format: %d:%d\n", hours, minutes);
}
```

17

Don't Overuse Call by Reference

```
int hours(int time)
{
   return time / MinutesPerHour;
}
int minutes(int time)
{
   return time % MinutesPerHour;
}
main()
{
   int time;
   scanf("%d", &time);
   printf("HH:MM format: %d:%d\n", hours(time), minutes(time));
}
```

18

Pointers and Arrays

- Pointers can also point to elements of an array

```
int array[10], *p;
p = &array[0];
*p = 10;
printf("%d, %d\n", array[0], *p);
```

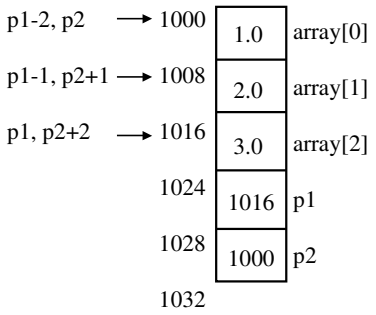
19

Pointer Arithmetic

- If a pointer points to elements of an array, some simple pointer arithmetic is meaningful
- If *p* points to array[*i*], *p*+*k* points to array[*i*+*k*]
- If *p* points to array[*i*], *p*-*k* points to array[*i*-*k*]
- If *p* points to array[*i*] and *q* points to array[*j*], *p*-*q* is equal to *i*-*j*

20

Pointer Arithmetic



21

An Example

```
main()
{
   int i, sum, array[10];

   for (i = 0; i < 10; i++) {
      scanf("%d", &array[i]);
   }
   sum = 0;
   for (i = 0; i < 10; i++) {
      sum += array[i];
   }
}
```

```
main()
{
   int i, sum, array[10], *p;

   for (i = 0; i < 10; i++) {
      scanf("%d", &array[i]);
   }
   sum = 0;
   for (p = &array[0]; p <= &array[9]; p++) {
      sum += *p;
   }
}
```

22

++ and --

- The postfix form: x++
  uses the value of x as the value of the expression first, and then increments it
- The prefix form: ++x
  increments the value of x first, and then uses the new value as the value of the expression

23

An Example

```
main()
{
   int x, y;

   x = 5;
   y = ++x;
   printf("x = %d, y = %d\n", x, y);
   x = 5;
   y = x++;
   printf("x = %d, y = %d\n", x, y);
}
```

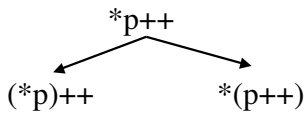
24

An Example

```
for (i = 0; i < n; i++)  arr[i] = 0;

for (i = 0; i < n;)  arr[i++] = 0;

for (i = 0; i < n;)  arr[i] = i++;
```

25

An Example



26

An Example

```
main()
{
   int i, sum, array[10], *p;

   for (i = 0; i < 10; i++) {
      scanf("%d", &array[i]);
   }
   sum = 0;
   for (p = array; p <= &array[9];) {
      sum += *p++;
   }
}
```

```
main()
{
   int i, sum, array[10];

   for (i = 0; i < 10; i++) {
      scanf("%d", array+i);
   }
   sum = 0;
   for (i = 0; i < 10; i++) {
      sum += *(array+i);
   }
}
```

27

An Example

```
int add(int array[], int size)
{
   int i, sum;
   sum = 0;
   for (i = 0; i < size; i++)
      sum += array[i];
   return sum;
}

main()
{
   int s, n[SIZE];
   s = add(n, SIZE);
}
```

≡

```
int add(int *array, int size)
{
   int i, sum;
   sum = 0;
   for (i = 0; i < size; i++)
      sum += *(array+i);
   return sum;
}

main()
{
   int s, n[SIZE];
   s = add(n, SIZE);
}
```

28

An Example

```
main()
{
   int i, sum, array[10];

   for (i = 0; i < 10; i++) {
      scanf("%d", &array[i]);
   }
   sum = 0;
   for (i = 0; i < 10; i++) {
      sum += array[i];
   }
   printf("%d\n", sum);
}
```

40 bytes

```
main()
{
   int i, sum, *array;

   for (i = 0; i < 10; i++) {
      scanf("%d", array+i);
   }
   sum = 0;
   for (i = 0; i < 10; i++) {
      sum += *(array+i);
   }
   printf("%d\n", sum);
}
```

4 bytes

error

error

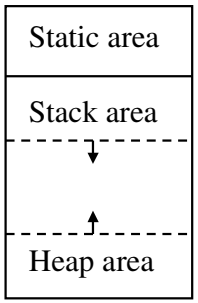
29

Dynamic Allocation

- *Static allocation*: memory spaces that are allocated in fixed locations and persist throughout the entire program
- *Automatic allocation*: memory spaces that are allocated when entering a function and freed when exiting a function
- *Dynamic allocation*: memory spaces that are explicitly allocated and freed by programmers while the program is running

30

Memory Organization



31

Malloc and Free

In stdlib.h:

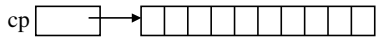
```
void *malloc(int nBytes);

void free(void *pointer);
```

void * is a *general* pointer type

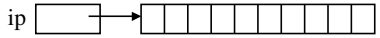
32

Malloc and Free

```
char  *cp;
cp = (char *) malloc(10 * sizeof(char));
free(cp);
```




```
int  *ip;
ip = (int *) malloc(10 * sizeof(int));
free(ip);
```



33

Dynamic Arrays

```
main()
{
   int i, sum, n, *array;

   scanf("%d", &n);
   array = (int *) malloc(n * sizeof(int));
   for (i = 0; i < n; i++)
      scanf("%d", array+i);   /* scanf("%d", &array[i]) */
   sum = 0;
   for (i = 0; i < n; i++)
      sum += *(array+i);      /* sum += array[i] */
   printf("%d\n", sum);
   free(array);
}
```

dynamic array

34

Detecting Errors in Malloc

```
main()
{
   int i, sum, n, *array;

   scanf("%d", &n);
   array = (int *) malloc(n * sizeof(int));
   if (array == NULL) { printf("Error: no more memory\n"); exit(1); }
   for (i = 0; i < n; i++)
      scanf("%d", array+i);
   sum = 0;
   for (i = 0; i < n; i++)
      sum += *(array+i);
   printf("%d\n", sum);
}
```

35