Day13 A

Young W. Lim

2017-10-25 Wed

Outline

1. Based on

2. Pointers (2) - Applications
   - Pointers and Arrays
   - Arrays of Pointers
   - Pointers to Functions
   - Using the const Qualifier with Pointers

Based on

"C How to Program",
Paul Deitel and Harvey Deitel

Pointers and 1-d Arrays

- are intimately related in C
- can be used interchangeably (`a[i] = *(a+i)`)
- the array name can be thought as a constant pointer
- pointers can do any operation that the array subscript does

Offset vs Subscript Notations (1)

- `int a[4];`
- `int *p = a;`

- `a` : 1-d array name
- `p` : pointer variable
- pointer-offset notation : `*(p+i)`, offset `i`
- pointer-subscript notation : `p[i]`, subscript `i`

Offset vs Subscript Notations (2)

- pointer-offset notation : `*(p+i)`
- pointer-subscript notation : `p[i]`

- when a pointer points to the beginning of an array
- by changing an offset which is added to this start pointer every element of the array can be referenced `*(p + i)`
- pointers can be subscripted exactly as arrays can `p[i]`

- the offset value is identical to the array subscript

1-d Array Names and Pointers (1)

- `int a[4]; int *p = a;`
- `a` : 1-d array name
- `p` : pointer variable

- a 1-d array name is a pointer but it is a constant pointer
- an 1-d array name cannot be modified
- `a[i]` subscript notation
- `*(a + i)` offset notation

1-d Array Names and Pointers (2)

```
int main(void) {
  int  a[4] = { 100, 200, 300, 400 };
  int *p;
  int i;

  for (i=0; i<4; ++i) {
    printf("&a[%d]= %p   ", i, &a[i]);
    printf(" a[%d]= %d \n", i,  a[i]);
  }

  p = a;
  for (i=0; i<4; ++i) {
    printf("p= %p   ", p);
    printf("*p= %d \n", *p);
    p += i;
  }
}
```

1-d Array Names and Pointers (3)

- `p += 1;` (OK - a pointer variable)
- `a += 1;` (Error - a constant pointer)

```
&a[0]= 0x7ffeac902c80    a[0]= 100
&a[1]= 0x7ffeac902c84    a[1]= 200
&a[2]= 0x7ffeac902c88    a[2]= 300
&a[3]= 0x7ffeac902c8c    a[3]= 400
p= 0x7ffeac902c80   *p= 100
p= 0x7ffeac902c80   *p= 100
p= 0x7ffeac902c84   *p= 200
p= 0x7ffeac902c8c   *p= 400
```

1-d Array Names and Pointers (4)

- the 1-d array name can be treated as a pointer
- but the array name pointer cannot be modified
- cannot do pointer arithemtic expressions
  that modify the pointer itself

Arrays of Pointers (1)

- arrays can contain pointers

```
int main(void) {
  int a     =   100;
  int A[4]  = { 100, 200, 300, 400 };

  int *p    =   &a;
  int *P[4] = { &A[0], &A[1], &A[2], &A[3] };

  int i;

  printf("----------------------\n");
  printf("&a= %p  ", &a);
  printf(" a= %d\n",  a);

  printf("----------------------\n");
  printf("&p= %p  ", &p);
  printf(" p= %p  ",  p);
  printf("*p= %d\n", *p);
```

Arrays of Pointers (2)

```
  printf("----------------------\n");
  for (i=0; i<4; ++i) {
    printf("&A[%d]= %p  ", i, &A[i]);
    printf(" A[%d]= %d\n", i,  A[i]);
  }

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

Arrays of Pointers (3)

- arrays can contain pointers

```
-----------------------
&a= 0x7ffc419f6a70   a= 100
-----------------------
&p= 0x7ffc419f6a78   p= 0x7ffc419f6a70  *p= 100
-----------------------
&A[0]= 0x7ffc419f6a80   A[0]= 100
&A[1]= 0x7ffc419f6a84   A[1]= 200
&A[2]= 0x7ffc419f6a88   A[2]= 300
&A[3]= 0x7ffc419f6a8c   A[3]= 400
-----------------------
&P[0]= 0x7ffc419f6a90   P[0]= 0x7ffc419f6a80  *P[0]= 100
&P[1]= 0x7ffc419f6a98   P[1]= 0x7ffc419f6a84  *P[1]= 200
&P[2]= 0x7ffc419f6aa0   P[2]= 0x7ffc419f6a88  *P[2]= 300
&P[3]= 0x7ffc419f6aa8   P[3]= 0x7ffc419f6a8c  *P[3]= 400
```

Arrays of Strings (1)

- an array of pointers can be used to form an array of strings
  - each entry in the array is a string
  - a string is a sequence of characters
    - the first character of a string is denoted by a pointer to that character
    - the end of a string is denoted by the null terminating character (~'\0'~)
  - a string is therefore identified by a pointer to its first character
  - each entry in an array of strings is such a pointer
    to the first character of a string

Arrays of Strings (2)

```
include <stdio.h>

int main(void) {
  char *s1 = "John";
  char *s2 = "Baker";
  char *s3 = "Park";
  char *s4 = "Kim";

  char *S[4] = { s1, s2, s3, s4 };
  int i;

  printf("s1= %s \n", s1);
  printf("s2= %s \n", s2);
  printf("s3= %s \n", s3);
  printf("s4= %s \n", s4);

  for (i=0; i<4; ++i) {
    printf("S[%d]= %s \n", i, S[i]);
  }

}
```

```
s1= John
s2= Baker
s3= Park
s4= Kim
S[0]= John
S[1]= Baker
S[2]= Park
S[3]= Kim
```

Pointer to a Function

- a pointer to a function contains the address of the function in memory
- a function name is really the starting address
  of the functions machine code that performs the function's task
- a pointer to a function is dereferenced (`*fp`)to call the function
- a function pointer (`fp`) can be used directly
  like the function name is used
  when calling the function
- a common use of function pointers is in text-based, menu-driven system

Function Pointer Examples (1)

```
#include <stdio.h>

int add(int x, int y) {
  return (x + y);
}

int sub(int x, int y) {
  return (x - y);
}

int mul(int x, int y) {
  return (x * y);
}

int div(int x, int y) {
  return (x / y);
}

int main(void) {
   int a = 30, b = 2;
   int v;

   int (*fp)(int x, int y);

   printf("a= %d   b= %d \n", a, b);

   printf("-----------------------\n");
   fp = &add;   v = (*fp)(a, b);
   printf("fp = &add  %p \n", fp);
   printf("(*fp)(a, b) = %d \n", v);

   printf("-----------------------\n");
   fp = &sub;   v = (*fp)(a, b);
   printf("fp = &sub  %p \n", fp);
   printf("(*fp)(a, b) = %d \n", v);

   printf("-----------------------\n");
   fp = &mul;   v = (*fp)(a, b);
   printf("fp = &mul  %p \n", fp);
   printf("(*fp)(a, b) = %d \n", v);

   printf("-----------------------\n");
   fp = &div;   v = (*fp)(a, b);
   printf("fp = &div  %p \n", fp);
   printf("(*fp)(a, b) = %d \n", v);

}
```

Function Pointer Examples (2)

```
a= 30   b= 2
------------------------
fp = &add  0x400596
(*fp)(a, b) = 32
------------------------
fp = &sub  0x4005aa
(*fp)(a, b) = 28
------------------------
fp = &mul  0x4005bc
(*fp)(a, b) = 60
------------------------
fp = &div  0x4005cf
(*fp)(a, b) = 15
```

- these are also working

```
fp = add;
v = (fp)(a, b);

fp = sub;
v = (fp)(a, b);

fp = mul;
v = (fp)(a, b);

fp = div;
v = (fp)(a, b);
```

const qualifier

- to indicate that the value of a particular variable should not be modified
- if an attempt is made to modify the value of a const quailifed variable either a warning or an error message will be issued
- four ways to pass a pointer to a function

a non-constant pointer to non-constant data	int *p
constant pointer to non-constant data	int * const p
a non-constant pointer to constant data	const int *p
a constant pointer to constant data	const int * const p

const qualifier usage

a non-constant pointer to non-constant data - the data can be modified through * - the pointer can point to other data	int *p
constant pointer to non-constant data - the data can be modified through * - the pointer cannot point to other data	int * const p array name
a non-constant pointer to constant data - the data cannot be modified through * - the pointer can point to other data	const int *p
a constant pointer to constant data - the data cannot be modified through * - the pointer cannot point to other data	const int * const p

const int * = int const * = const int const *