Advanced C Features

by NZP

What you will learn

- Defining Functions
- Recursion
- Pointers
- Dynamic Memory Allocation
- Pointers to Functions

Function

- Motivations:
  - Software design experience: The best way to develop algorithms is to break up software into modules: Divide and conquer!
  - Software reusability: using existing functions to create new programs
  - To avoid repeating code in a program.

Functions

- All variables declared in function definitions are local variables: They are known only in the function they are defined.
- Most functions have a list of parameters, which are also local variables.

Function Definitions

- The format of a function definition is:

```
return-value-type function-name(parameter-list)
{
   declarations;
   statements;
}
```

Function prototype

- Function prototype declares a function before it is used and prior to its definition.
- Consists of
  - a function's name,
  - its return type
  - its parameter list
- The compiler needs to know this information

Example: Programmer defined function maximum to determine and return the largest of three integers

```c
#include <stdio.h>
int maximum(int x, int y, int z); /* function prototype */

int main(void) {
   int a, b, c;
   printf("Enter three integers:");
   scanf("%d%d%d",&a,&b,&c);
    /* function call */
   printf("Maximum is: %d\n", maximum(a, b, c));
    return 0;
}


/* Function definition */
int maximum( int x, int y, int z)
{
   int max = x;

   if ( y > max ) max = y;
   if ( z > max ) max = z;

   return max;
}
```

Passing Arrays to Functions

- Variables are passed to functions call-by-value: Changes made to a parameter of the function have no effect on the argument used to call it.

- Arrays are passed to functions call-by-reference: The address of an argument into the parameter, thus changes made to the parameter will affect the argument.
  Why: Array name is the same as the address of the array's first element!

Example: Modify values of array elements

```c
#include <stdio.h>
#define SIZE 5

void modifyArray( int [], int );

int main(void) {
   int a[ SIZE ] = {0, 1, 2, 3, 4 };
   int i;

   modifyArray( a, SIZE );

   /* output modified array */
   for( i = 0; i < SIZE; i++ ) {
       printf("%3d", a[i]);
   }
   printf("\n");

    return 0;
}

/* Function definition */
void modifyArray( int b[], int size )
{
   int j;

   for ( j = 2; j < size; ++j ) {
       b[ j ] *= 2;
   }

}
```

Recursion

- Recursion is the process by which something is defined in terms of itself. (GNU's not UNIX)
- In C a function can call itself!

Example: Calculation of n! by recursive calls

n! = n (n-1)! is the recursive definition

```c
#include <stdio.h>
long factorial(int);

int main(void) {
   int i;

   for( i = 0; i <= 10; i++ ) {
       printf("%2d!= %ld\n", i, factorial(i));
   }

    return 0;
}

/* Function definition */
long factorial( int number )
{
   if (number <= 1) {
        return 1;
   }
   else {
        return ( number * factorial(number - 1));
   }

}
```

Recursion

- The function has two parts: the base case part that returns 1, and recursive call part.
- Omitting the base, or writing recursion step incorrectly causes infinite recursion!
- Each recursive call creates a new copy of the function! Thus recursive calls take time and consume additional memory.

Pointers

- Definition: A variable whose value is a memory address
- Declaration: Precede the variable name by *
  int *P; /* P is a pointer to int */
- Pointers can be defined to point to objects of any data type.
- A pointer may be initialized to 0, NULL or an address. 0 or NULL means "points to nothing"

Pointer Operator &

- & is the address operator

```
int y = 5;
int *yPtr;
yPtr = &y; /* assigns the address of the
              variable y to pointer variable yPtr */
```

Pointer Operator *

- * is the indirection operator, it returns the value of the variable that the pointer points to.

```
int y = 5;
int *yPtr;
yPtr = &y;
printf( "%d", *yPtr ); /* prints the value of
                          variable y, namely 5 */
```

Example: Square a variable using call-by-reference

```c
#include <stdio.h>
void squareByReference(int *);

int main(void) {
  int number = 5;

  squareByReference( &number );
  printf("%d\n",number);

    return 0;
}

/* Function definition */
void squareByReference( int *n )
{
  *n = *n * *n;
}
```

Pointer Arithmetic

- A pointer may be incremented (++)
- A pointer may be decremented (--)
- An integer may be added to a pointer (+ or +=)
- An integer may be subtracted from a pointer (- or -=)
- But this is no ordinary arithmetic unless the pointer pointes to a variable of type char!

Pointer Arithmetic

Assume that array int v[5] has been defined and is at location 3000 in memory. Assume pointer vPtr has been initialized to point to v[0]. Recall that integers are 4 bytes. The initialization can be done in two ways:

vPtr = v; or vPtr = &v[0];

When an integer is added or subtracted from a pointer, the pointer is incremented or decremented by that integer times the size of the object to which the pointer refers. For example,

vPtr += 2;

would produce 3008 (3000 + 2 * 4).

Example: Relation between arrays and pointers

```c
#include <stdio.h>

int main(void) {
  char array[5];

  printf(" array = %p\n"
         " &array[0] = %p\n"
         " &array = %p\n",
     array, &array[0], &array);

  return 0;
}
```

Output

```
array = 0x7fff5fbff950
&array[0] = 0x7fff5fbff950
&array = 0x7fff5fbff950
```

Explanation: Array name is the same as the address of the array's first element!

%p is used for printing addresses. It outputs addresses as hex numbers.

Dynamic Memory Allocation

- Dynamic allocation is the process by which memory is allocated as needed during runtime.
- C's dynamic-allocation functions are
  - malloc() allocates memory
  - free() releases previously allocated memory

Example: Create a dynamic character array.
Use the allocated memory to input a string.

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

int main(void) {
   char *p;

   p = malloc(80); /* allocate 80 bytes and a character pointer
                      to it */

   if (!p) {
       printf("Allocation Failed");
       exit(1);
   }

   printf("Enter a string: ");
   gets(p);
   printf(p);
   free(p);

   return 0;
}
```

Function Pointers

- A function pointer is a variable that contains the address of the entry point to a function.
- Once you have a pointer to a function it is possible to call that function using the pointer!
- int (*p) (int x, int y); declares p as a pointer to a function that returns an integer and has two integer parameters x and y. Be sure to include parentheses surrounding *p.

Function Pointers

- int (*p) (int x, int y);
- Assume that sum() has the prototype:
  int sum(int, int);
  the assignment statement: p = sum;
  is correct!
- Can call sum() indirectly through p like:
  result = (*p) (10, 20);
  or
  result = p(10, 20); /* should not be preferred */