/* EXAMPLE 3 */
#include <stdio.h>
int main(void) {
    char ch = 'c';
    char *chptr = &ch;
    int i = 20;
    int *intptr = &i;
    float f = 1.20000;
    float *fptr = &f;
    char *ptr = "I am a string";
    printf("\n [%c], [%d], [%f], [%c], [%s]\n", *chptr, *intptr, *fptr, *ptr, ptr);
    return 0; }

/* EXAMPLE 1 */
#include<stdio.h>
int main() {
    float i=10, *j;
    void *k;
    k=&i;
    j=k;
    printf("%f\n", *j);
    return 0; }

/* EXAMPLE 2 */
#include <stdio.h>
#include <stdlib.h>
main() {
    int x, *p;
    p = &x;
    *p = 0;
    printf("x is %d\n", x);
    printf("*p is %d\n", *p);
    *p += 1;
    printf("x is %d\n", x);
    (*p)++;
    printf("x is %d\n", x);
    return 0; }

You try…
Function definition

- Function prototype
  - Return type
  - Argument definition
  - return_type function_name (type1 arg1,type2 arg2,..,typen argn)

- Function calls
  - Basic syntax
  - Parameter passing

- Standard library and function calls

NOTES:
- Functions should be short and sweet.
- Functions do not nest.
- Variables have to be communicated through function arguments or global variables.
- * If no return type or no argument type, then it defaults to int

Pointers and Functions

Pass By Value
- Passing a variable by value makes a copy of the variable before passing it onto a function. This means that if you try to modify the value inside a function, it will only have the modified value inside that function. Once the function returns, the variable you passed it will have the same value it had before you passed it into the function.

Pass By Reference
- There are two instances where a variable is passed by reference:
  - When you modify the value of the passed variable locally (inside the callee) and the value of the variable in the calling function as well.
  - To avoid making a copy of the variable for efficiency reasons.
A number of statements grouped into a single logical unit are called a function.

REMINDER ➔ It is necessary to have a single function ‘main’ in every C program.

A function prototype is a function declaration or definition which includes:

- Information about the number of arguments
- Information about the types of the arguments

Although you are allowed not to specify any information about a function's arguments in a declaration, it is purely because of backwards compatibility with Old C and should be avoided (poor coding style).

A declaration without any information about the arguments is not a prototype.

Only one function with a given name may be defined. Unlike Java, C does not support overloading (i.e., two functions with the same name but different signatures).
Pointers and Function arguments

Since C passes arguments to functions by value and make a copy local to swap; so there is no direct way for the called function (callee) to alter a variable in the calling function (caller).

Because of call by value, swap can’t affect the arguments a and b in the routine that called it.

The way to obtain the desired effect is for the calling program to pass pointers to the values to be changed:

Since the operator & produces the address of a variable, &a is a pointer to a. In swap itself, the parameters are declared as pointers, and the operands are accessed indirectly through them.

NOW KNOW WHY SCANF NEEDS & SYMBOLS!!!
Every function except those returning void should have at least one, each return showing what value is supposed to be returned at that point.

Although it is possible to return from a function by falling through the last }, unless the function returns void, an unknown value will be returned, resulting in undefined behavior.

The type of expression returned must match the type of the function, or be capable of being converted to it as if an assignment statement were in use.

Following the return keyword with an expression is *not* permitted if the function returns void.
Another Function example

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

main ()                       /* was-could be void pmax(); now considered bad practice */
{  void pmax(int first, int second);                     /* declaration prototype */
  int i,j;
  for(i = -10; i <= 10; i++)
    { for(j = -10; j <= 10; j++)
      { pmax(i,j);
        }
    }
  return 0;
}

/* Prints larger of its two arguments. */
void pmax (int a1, int a2)                               /* definition */
{   int biggest;
  if (a1 > a2)
    { biggest = a1; }
  else { biggest = a2; }
  printf("larger of %d and %d is %d\n", a1, a2, biggest);
  }
```
#include <stdio.h>
#include <stdlib.h>

void printtotal(int total);
void addxy(int x, int y, int total);
void subxy(int x, int y, int *total);

void main() {
    int x, y, total;
    x = 10;
    y = 5;
    total = 0;
    printtotal(total);
    addxy(x, y, total);
    printtotal(total);
    subxy(x, y, &total);
    printtotal(total); }

void printtotal(int total) {
    printf("Total in Main: %dn", total);
}

void addxy(int x, int y, int total) {
    total = x + y;
    printf("Total from inside addxy: %dn", total);
}

void subxy(int x, int y, int *total) {
    *total = x - y;
    printf("Total from inside subxy: %dn", *total);
}
Another Function example

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

void date(int *, int *);  /* declare the function */

main() {
    int month, day;
    date(&day, &month);
    printf("day is %d, month is %d\n", day, month);
    return 0;}

void date(int *day_p, int *month_p) {
    int day_ret, month_ret;
    /* * At this point, calculate the day and month * 
    values in day_ret and month_ret respectively. */
    *day_p = day_ret;
    *month_p = month_ret; }
```
Function Summary

- Functions can be called recursively.
- Functions can return any type that you can declare, except for arrays and functions (you can get around that restriction to some extent by using pointers).
- Functions returning no value should return void.
- Always use function prototypes.
- Undefined behavior results if you call or define a function anywhere in a program unless either
  - a prototype is always in scope for every call or definition, or
  - you are very, very careful.
- Assuming that you are using prototypes, the values of the arguments to a function call are converted to the types of the formal parameters exactly as if they had been assigned using the = operator.
- Functions taking no arguments should have a prototype with (void) as the argument specification.