

CS240 Notes

Maninder (Kaurman) Kaur

July 14, 2025

Week 1

Version 3

```
/* version 3 of z = x * y
   reads the numbers to be subtracted from keyboard
   using the standard I/O library function scanf()
   and outputs the result on the terminal
   using printf() */
```

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

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

    // read input
    scanf("%d-%d", &x, &y);

    /* compute multiplication */
    z = x * y;

    // print result
    printf("%d-*-%d=-%d\n", x, y, z);
}
```

`main()` calls `scanf()` to do something for it; the two inputs that should be read from the user should be stored into `int x` and `y`. This is done by putting every function from MAIN MEMORY, where they get their own working area. It is allocated for the function to use, allowing `main()` to call and use `scanf()`. Passing functions means to use them.

Alice and Bob are friends. She writes him two letters, placing them in mailbox 5 and 7 at the UPS office. Bob comes in later and opens 5 and 7 for

each letter. Alice and Bob represent the main function and the memory, while the letters are the functions.

Imagine memory as a bunch of slots that allow you to place data like bytes. Each slot allows 8 bits. The memory slots start at index 0 and go up to $2^n - 1$ slots. Integers take up 4 bytes.

How is this different than printf?

`main()` calls `printf()` to print on the terminal. It will print just the input of the variable. There is no need to store anything. `scanf` needs to know the address, while `printf` does not.

Segmentation Fault

A segmentation fault occurs when you try to access a data value that the OS does not give access to.

Version 4

```
/*      version 4 of z = x * y
       same as version 3 but supports
       real numbers */

#include <stdio.h>

int main()
{
    float x, y, z;

    // read input
    scanf("%f%f", &x, &y);

    // multiply
    z = x * y;

    // print result
    printf("result of %f times %f is %f\n", x, y, z);
}
```

Version 5

```
/*      version 5 of z = x * y
       same as version 4 but uses separate
       function multiply2() to perform multiplication */
```

```

#include <stdio.h>

float multiply2(float, float);

void main()
{
    float x, y, z;

    // read input
    scanf("%f%f", &x, &y);

    // compute
    z = multiply2(x, y);

    // print result
    printf("result of %f * %f is %.3f\n", x, y, z);
}

/*      function multiply2(a,b) takes two
        arguments of type float, multiplies a
        and b, and returns the result to
        the calling function */

float multiply2(float a, float b)
{
    float c;

    // multiply a with b
    // and store the result in local variable c
    c = a * b;

    // return value of c to calling function
    return c;
}

```

`printf()` works as follows: if there is a variable `x` and we assign it a value, to print it we would simply use `printf("%d", x)`. However, with `scanf`, we would use `scanf("%d", &x)`. We use `&` because we are not passing the value of `x`, but using the memory address itself to store the value.

Week 2

Linking and Loading

When we have a function like `multiply2()`. GCC will link the function statically, means that after being translated into machine code, it will be integrated into `a.out`. I called from these files `printf()` and `scanf()`, both pre-written code, which is an example of how often code for others and not just ourselves. The code we make is deposited into a library to be used which is used in the process of Linking and Loading.

We will typically dynamically link. Lets use the example that we use `borg02`, the server will share all the machine code from some library, which reduces the memory consumption. One copy of a function in a library in the `usr/` directories.

Loading is about loading a code segment to allow it to become an executable and loaded into main memory.

Version 1

```
// Program to illustrate content vs. address  
// of a local variable.
```

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

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

```
    x = 7;  
    printf("%d\n",x);
```

```
    // format %p is for printing address  
    printf("%p\n",&x);
```

```
}
```

When running `a.out` of Version 1, the output integer is printed and then the address of where it is held. `0x` indicates that the address is hexadecimal which allows 4-bit systems. Each slot is given a memory address. We use the numbers 0-9 and letters A-F. (i.e. Look at hexadecimal computer conversion).

Each byte outputs 4 bits, and there are 12 bytes after `0x`, indicating there will be 48-bits will be outputted in total.

Version 2

```
// Meaning of a pointer: a variable whose content is an address.
```

```

#include <stdio.h>

int main()
{
    int x, *y;

    x = 7;
    printf("%d\n", x);
    printf("%p\n", &x);

    y = &x;
    printf("%d\n", *y);
    printf("%p\n", y);

    // meaning of int **z?
}

```

Version 3

// Meaning of a pointer: a variable whose content is an address.

```

#include <stdio.h>

int main()
{
    int x, *y;

    x = 7;
    printf("%d\n", x);
    printf("%p\n", &x);

    y = &x;
    printf("%d\n", *y);
    printf("%p\n", y);

    // meaning of int **z?
}

```

Version 4

*// Use functions changeling1() and changeling2() to illustrate
// passing by value vs. reference (i.e., address).*

```

#include <stdio.h>

```

```

void changeling1(int);
void changeling2(int *);

```

```

int main()
{
int r;

    r = 7;
    changeling1(r);
    printf("%d\n", r);

    r = 9;
    changeling2(&r);
    printf("%d\n", r);
}

```

```

void changeling1(int x)
{
    x = 100;
}

```

```

void changeling2(int *y)
{
    *y = 200;
}

```

Version 7

// Pointers and arrays: 1-D array.

```

#include <stdio.h>

```

```

int main()
{
int z[5];

    z[0] = 100;
    z[1] = 200;
    z[2] = 300;
    z[3] = 400;
    z[4] = 500;
}

```

```

printf("%d\n", z[0]);
printf("%d\n", z[1]);
printf("%d\n", z[2]);
printf("%d\n", z[3]);
printf("%d\n", z[4]);

printf("%d\n", *z);
printf("%d\n", *(z+1));
printf("%d\n", *(z+2));
printf("%d\n", *(z+3));
printf("%d\n", *(z+4));

*z = 1;
*(z+1) = 2;
*(z+2) = 3;
*(z+3) = 4;
*(z+4) = 5;

printf("%d\n", z[0]);
printf("%d\n", z[1]);
printf("%d\n", z[2]);
printf("%d\n", z[3]);
printf("%d\n", z[4]);

}

```

Unlike making several different int values a,b,c, we make an array called z[3] where all the values are adjacent in the memory block. If z[1] is at address 100, the next int value z[2] will be at address 104, because int values are of 4 bytes.

This 1-D array allows for random access which means that it makes the memory slots adjacent. The only memory address we need to remember is the slot of index 0 (the starting).

To access the value of z[0], we should call *z. It will automatically start at the starting location. If we wanted to access z[2], we would use *(x+2) (or *(x+8) because 2 ints is 4*2 bytes).

Version 8

// Pointers and arrays: valid vs. invalid memory.

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

```
int main()
{
```

```

int *z;

*z = 100;
*(z+1) = 200;
*(z+2) = 300;

printf("%d\n",*z);
printf("%d\n",*(z+1));
printf("%d\n",*(z+2));

}

```

Notice how we don't have a definitive idea of where *z is pointing at. Because we don't know where the address is, it will probably end in a segmentation fault.

Version 9

```

// Pointers, memory, and silent run-time errors.

#include <stdio.h>

int main()
{
int s[5];
int i;

    for (i=0; i<5; i++)
        s[i] = i;

    for (i=0; i<5; i++)
        printf("%d\n",s[i]);

    // doing something sketchy
    for (i=0; i<6; i++)
        s[i] = i;

    for (i=0; i<6; i++)
        printf("%d\n",s[i]);

}

```

Notice how there are only 5 slots in s[]. However, we are asking to go past that threshold in the for loops. This does not cause a crash, but practices such as these should not be normalized. These are called "silent errors". However, if we increase the max count from 6 to something higher, it will crash and give us a message saying "stack smashing detected; Terminated Abort". This is

called **Stack Overflow**, because the function attempts to access outside of the memory slots allocated for the array.

Let's use the example of `main()` and `multiply2()`. If the functions are in the same file, then the memory allocated for them in the stack will be next to one another. Each of the functions hold the specific memory slots hold the following things:

1. Local Variables
2. Arguments
3. Return Addresses

A **canary** is a special bit address that is placed around a return addresses to stop malicious overriding. If the canary is the same from before, it will run, otherwise if compromised, it will refuse to run due to stack smashing.

Despite being helpful, canaries do slow down the program. If you run `gcc -fno-stack-protector main.c`, it will override the stack smashing error.

Version 10

```
// Basic string processing.
```

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

```
int main()
```

```
{
```

```
char x[10];
```

```
int i;
```

```
    x[0] = 'P';
```

```
    x[1] = 'u';
```

```
    x[2] = 'r';
```

```
    x[3] = 'd';
```

```
    x[4] = 'u';
```

```
    x[5] = 'e';
```

```
    x[6] = '\0';
```

```
    printf("%s\n", x);
```

```
}
```

The `\0` character is called the "end of string" or simply, EOS. All strings end with the EOS and the EOS lets the computer know that its a string.

Week 3

Version 5

// Another passing by address/reference example.

```
#include <stdio.h>

void changeval(int *);

int main()
{
    int s;

    s = 3;

    printf("inside main():\n");
    printf("value of s=%d\n", s);
    printf("address of s=%p\n", &s);
    printf("\n");

    // jump to changeval()
    changeval(&s);

    printf("back to main():\n");
    printf("value of s=%d\n", s);
    printf("address of s=%p\n", &s);
}

void changeval(int *t)
{
    printf("inside changeval():\n");

    // where does argument t live?
    printf("address of argument t=%p\n", &t);

    // what is the value of argument t?
    printf("value of argument t=%p\n", t);
    printf("\n");

    *t = 99;
}
```

Week 4

If the most significant bit is 0 in a 8 bit (byte), it is a ASCII character. We humans use ASCII to visably use, such as the words on our keyboard. If 0 is the first bit, there are 7 other slots for a 1 or 0 to be given. Thus, there is $2^7 = 128$ possible combinations. If the bit is over the 127 permutations (cause we start at index 0), then it will have a most significant bit starting with 1.

1 = Non-ASCII, 0 = ASCII and this logic works in all files.

TIP: use `% man (2 or 3 or neither) {inset function of command}` to get all the documentation from the terminal since we have downloaded C. :

main4.c

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

int main() {
    FILE *fp;

    fp = fopen("abcv2.dat", "w");
    if (fp == NULL) {
        fprintf(stderr, "File could not be opened.\n");
        exit(1);
    }

    fputc('a', fp);
    fputc('b', fp);
    fputc('c', fp);
}
```

The above creates a file object and stores the contents of abcv2.dat into it. The w mode allows for writing. However, if there is no file or it is corrupted, it will return an error message.

After that, it will place abc into the file. It appends to the end of file, that is why it goes into the line abc.

```
fputc("[content to append]", [file to be placed in])
```

main5.c

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

int main() {
    FILE *fp;
```

```

char filename[100];
int x, count;

scanf("%99s", filename);

fp = fopen(filename, "r");
if(fp == NULL) {
    fprintf(stderr, "File could not be opened.\n");
    exit(1);
}

count = 0;
while((x = fgetc(fp)) != EOF)
    count++;

printf("Size in bytes: %d\n", count);
}

```

In this code, we count each character. Starting by loading up `fp` with a file, we iterate in a while loop until the file ends with EOF. For each character, we add a number to our counter.

Week 5

A left shift if `0 << 1` will always introduce a 0 in the front. ANDing a bit often "squashes" it and makes it 0.