C Programming Language

Prof. Hyuk-Yoon Kwon
http://bigdata.seoultech.ac.kr

Most parts are based on slides used in Washington University by Fred Kuhns
(http://discovery.csc.ncsu.edu/Courses/csc405-F06/wrap/kuhns-wustl-summaryofc.ppt)

Contents

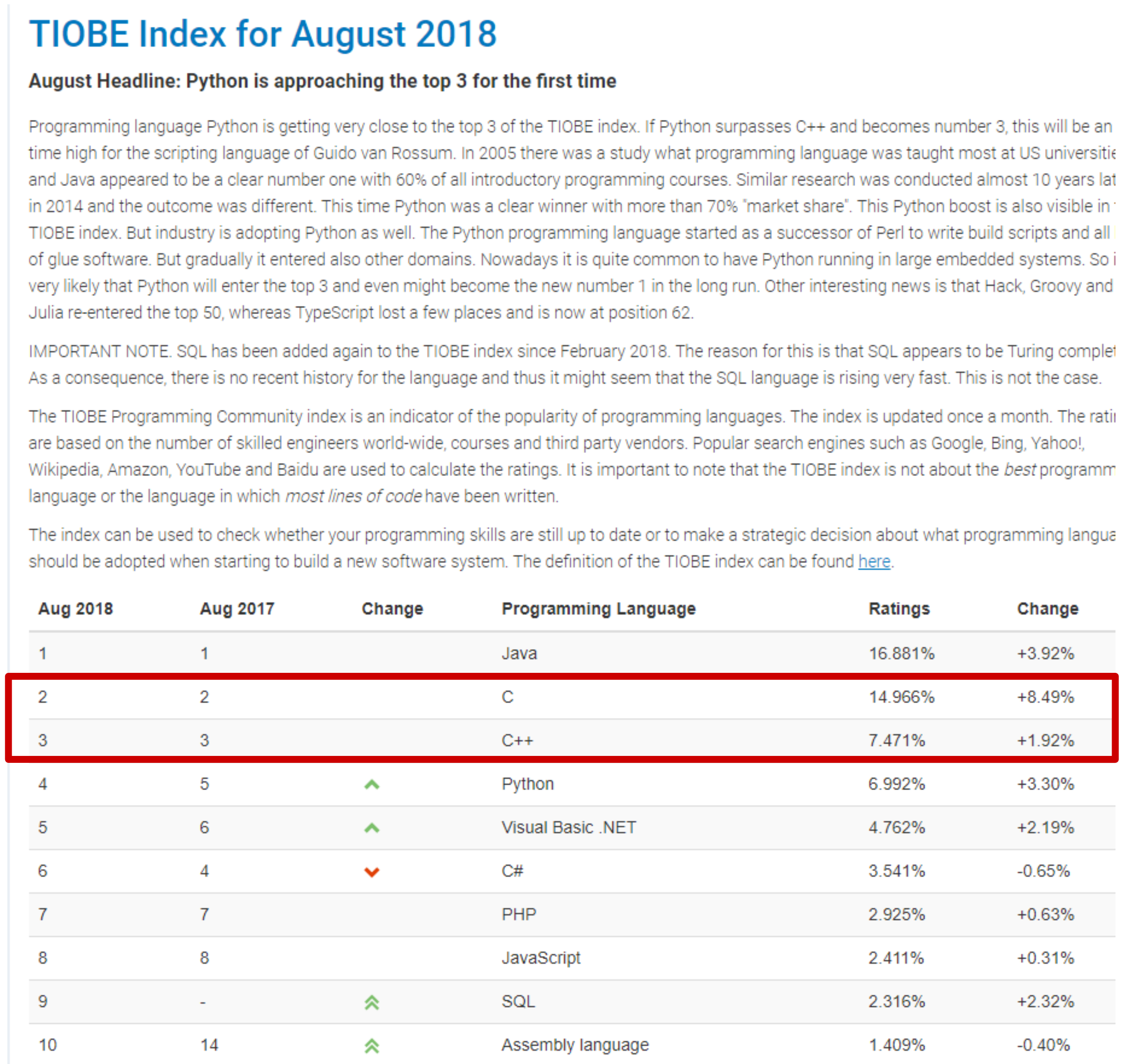
Introduction

- **The C programming language was designed by Dennis Ritchie at Bell Laboratories in the early 1970s**
- **Influenced by**
  - ALGOL 60 (1960),
  - CPL (Cambridge, 1963),
  - BCPL (Martin Richard, 1967),
  - B (Ken Thompson, 1970)
- **Traditionally used for systems programming, though this may be changing in favor of C++**
- **Traditional C**
  - *The C Programming Language*, by Brian Kernighan and Dennis Ritchie, 2nd Edition, Prentice Hall

Standard C

- **Standardized in 1989 by ANSI (American National Standards Institute) known as ANSI C**
- **International standard (ISO) in 1990 which was adopted by ANSI and is known as *C89***
- **As part of the normal evolution process the standard was updated in 1995 (*C95*) and 1999 (*C99*)**
- **C++ and C**
  - C++ extends C to include support for Object Oriented Programming and other features that facilitate large software development projects
  - C is not strictly a subset of C++, but it is possible to write "*Clean C*" that conforms to both the C++ and C standards.

Why C Programming Language?



TIOBE Index for August 2018

August Headline: Python is approaching the top 3 for the first time

Programming language Python is getting very close to the top 3 of the TIOBE index. If Python surpasses C++ and becomes number 3, this will be an time high for the scripting language of Guido van Rossum. In 2005 there was a study what programming language was taught most at US universitie and Java appeared to be a clear number one with 60% of all introductory programming courses. Similar research was conducted almost 10 years lat in 2014 and the outcome was different. This time Python was a clear winner with more than 70% "market share". This Python boost is also visible in TIOBE index. But industry is adopting Python as well. The Python programming language started as a successor of Perl to write build scripts and all of glue software. But gradually it entered also other domains. Nowadays it is quite common to have Python running in large embedded systems. So i very likely that Python will enter the top 3 and even might become the new number 1 in the long run. Other interesting news is that Hack, Groovy and Julia re-entered the top 50, whereas TypeScript lost a few places and is now at position 62.

IMPORTANT NOTE. SQL has been added again to the TIOBE index since February 2018. The reason for this is that SQL appears to be Turing complet As a consequence, there is no recent history for the language and thus it might seem that the SQL language is rising very fast. This is not the case.

The TIOBE Programming Community index is an indicator of the popularity of programming languages. The index is updated once a month. The ratii are based on the number of skilled engineers world-wide, courses and third party vendors. Popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings. It is important to note that the TIOBE index is not about the *best* programm language or the language in which *most lines of code* have been written.

The index can be used to check whether your programming skills are still up to date or to make a strategic decision about what programming langua should be adopted when starting to build a new software system. The definition of the TIOBE index can be found here.

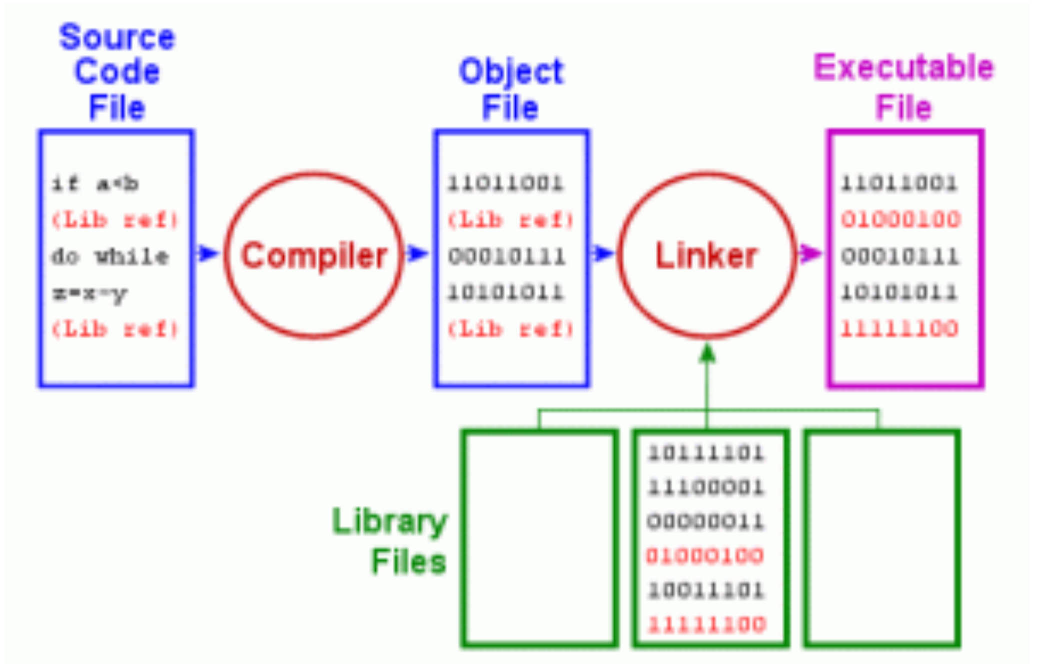
Aug 2018	Aug 2017	Change	Programming Language	Ratings	Change
1	1		Java	16.881%	+3.92%
2	2		C	14.966%	+8.49%
3	3		C++	7.471%	+1.92%
4	5	˄	Python	6.992%	+3.30%
5	6	˄	Visual Basic .NET	4.762%	+2.19%
6	4	˅	C#	3.541%	-0.65%
7	7		PHP	2.925%	+0.63%
8	8		JavaScript	2.411%	+0.31%
9	-	⏫	SQL	2.316%	+2.32%
10	14	⏫	Assembly language	1.409%	-0.40%

- C is widely used for system programming in implementing operating systems and embedded system applications

- C has both directly and indirectly influenced many later languages such as C#, D, Go, Java, JavaScript, Limbo, LPC , Perl, PHP, Python, and Unix's C shell

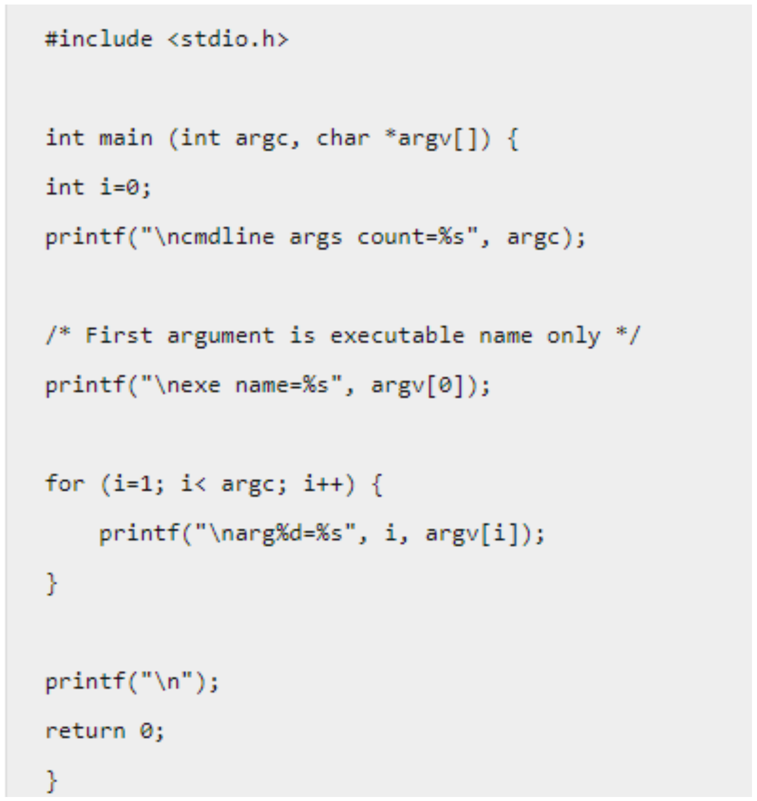
Elements of a C Program

- **A C development environment includes**
  - *System libraries* and *headers*: a set of standard libraries and their header files. For example, see `/usr/include` and `glibc`.
  - *Application Source*: application source and header files
  - *Compiler*: converts source to object code for a specific platform
  - *Linker*: resolves external references and produces the executable module

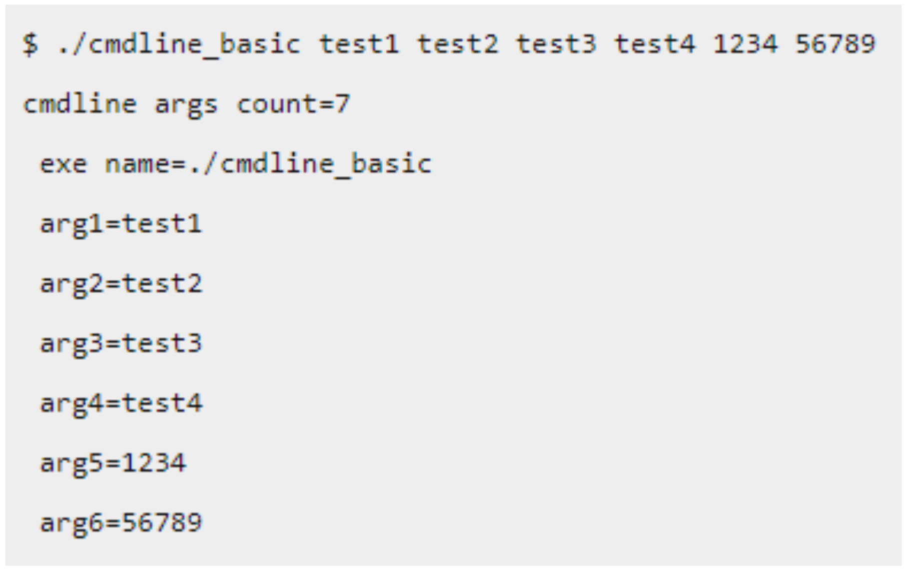


User Program Structure

- **There must be one main function where execution begins when the program is run. This function is called main**
  - `int main (void) { ... },`
  - `int main (int argc, char *argv[]) { ... }`



```c
#include <stdio.h>

int main (int argc, char *argv[]) {
int i=0;
printf("\ncmdline args count=%s", argc);

/* First argument is executable name only */
printf("\nexe name=%s", argv[0]);

for (i=1; i< argc; i++) {
    printf("\narg%d=%s", i, argv[i]);
}

printf("\n");
return 0;
}
```



```
$ ./cmdline_basic test1 test2 test3 test4 1234 56789
cmdline args count=7
 exe name=./cmdline_basic
 arg1=test1
 arg2=test2
 arg3=test3
 arg4=test4
 arg5=1234
 arg6=56789
```

A Simple C Program

- *Create* example file: `try.c`
- *Compile* using gcc:

```
gcc –o try try.c
```

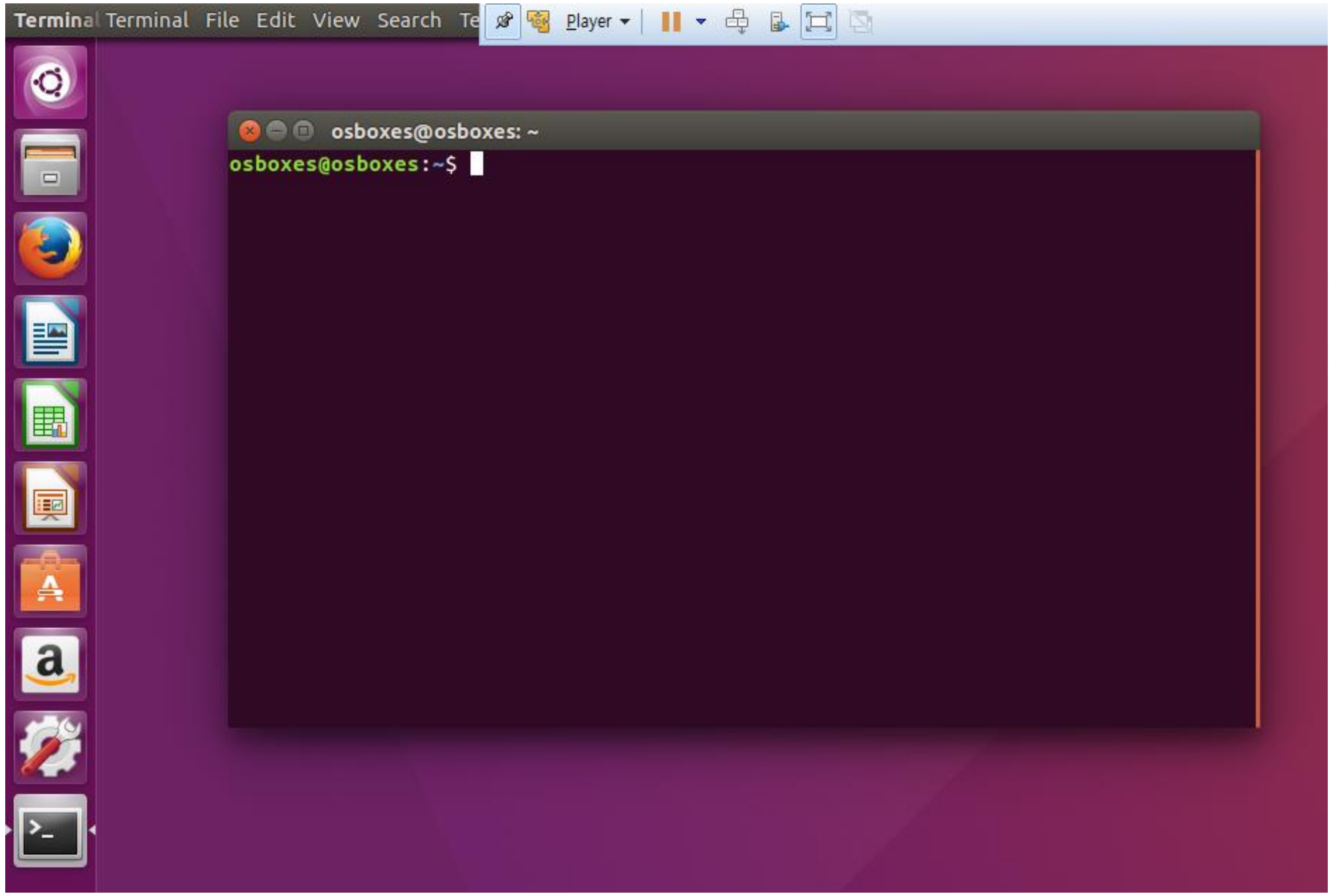
- The standard C library *libc* is included automatically
- *Execute* program

```
./try
```

```
/* you generally want to
 * include stdio.h and
 * stdlib.h
 * */
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
   printf(“Hello World\n”);
   exit(0);
}
```

Practice1: Simple C Program

- **Open the Terminal**
  - Shortcut: ctrl + alt + T

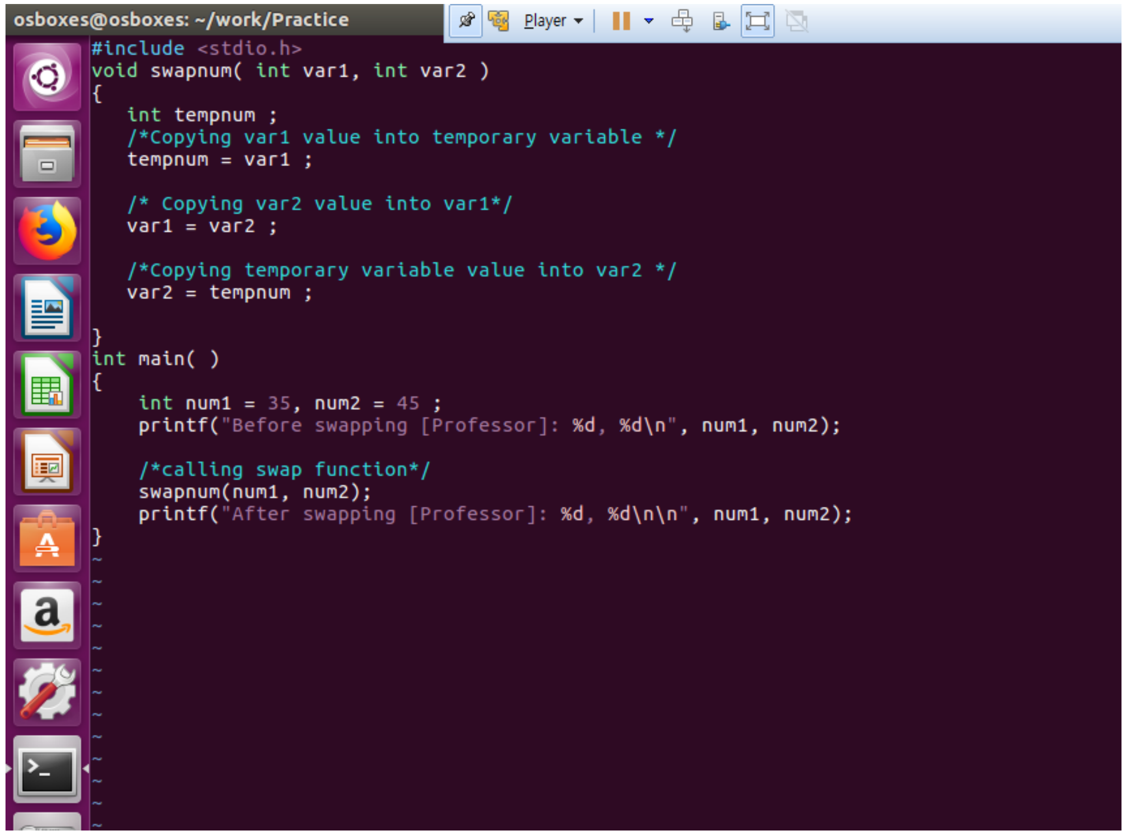


```
osboxes@osboxes:~$ 
```

Practice1: Simple C Program

- **Install some useful utilities**
  - VMTools
    - “Drag-And-Drop” and “Copy Files” from Windows
    - Execute following command “sudo apt-get install open-vm-tools”
  - VIM
    - Code editor (or other editors are possible, e.g., gedit)
    - “sudo apt-get install vim”

    - Write: input “i” then write codes
    - Store: input “ESC” then “:w”
    - Quit: input “ESE” then “:q”



osboxes@osboxes: ~/work/Practice

```c
#include <stdio.h>
void swapnum( int var1, int var2 )
{
   int tempnum ;
   /*Copying var1 value into temporary variable */
   tempnum = var1 ;

   /* Copying var2 value into var1*/
   var1 = var2 ;

   /*Copying temporary variable value into var2 */
   var2 = tempnum ;

}
int main( )
{
    int num1 = 35, num2 = 45 ;
    printf("Before swapping [Professor]: %d, %d\n", num1, num2);

    /*calling swap function*/
    swapnum(num1, num2);
    printf("After swapping [Professor]: %d, %d\n\n", num1, num2);
}
```

Source and Header files

- Place related codes within the same module (i.e., file)
- Header files (`*.h`) export interface definitions
  - function prototypes, data types, macros, inline functions and other common declarations
- Do not place source code (i.e., definitions) in the header file with a few exceptions
  - inline'd code
  - class definitions
  - const definitions

exceptions

- *C preprocessor* is used to insert common definitions into source files
- There are other cool things you can do with the preprocessor

Another C Program

`#include` directs the preprocessor to "include" the contents of the file at this point in the source file.
`#define` directs preprocessor to define macros.

/usr/include/stdio.h

```
/* comments */
#ifndef _STDIO_H
#define _STDIO_H

... definitions and protoypes

#endif
```

/usr/include/stdlib.h

```
/* prevents including file
 * contents multiple
 * times */
#ifndef _STDLIB_H
#define _STDLIB_H

... definitions and protoypes

#endif
```

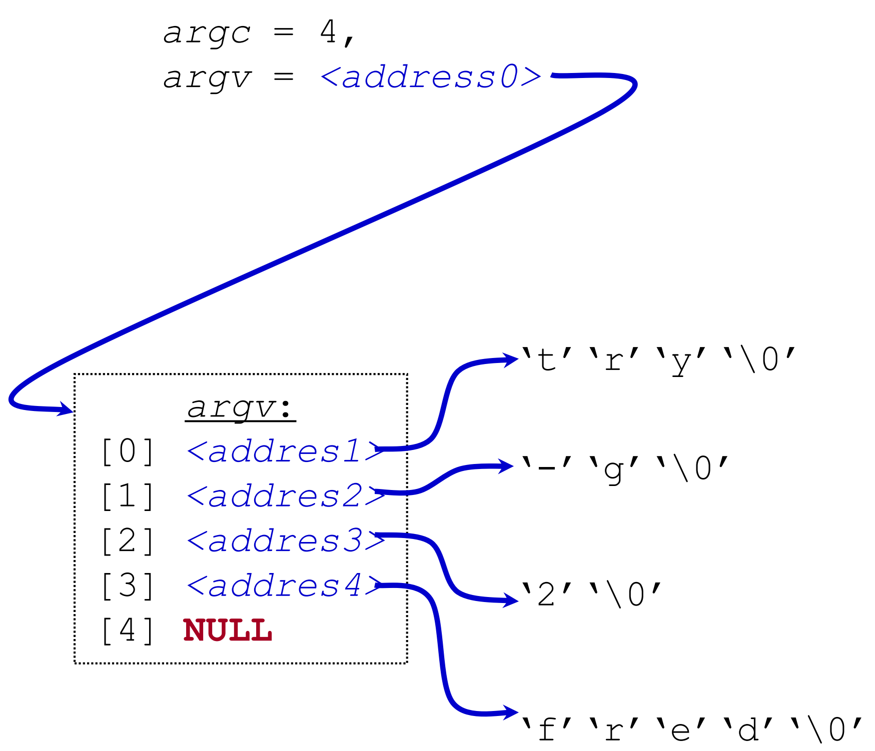
example.c

```
/* this is a C-style comment
 * You generally want to place
 * all file includes at start of file
 * */
#include <stdio.h>
#include <stdlib.h>

int
main (int argc, char **argv)
{
  // this is a C++-style comment
  // printf prototype in stdio.h
 printf("Hello, Prog name = %s\n",
             argv[0]);
 exit(0);
}
```

Passing Command Line Arguments

- When you execute a program you can include arguments on the command line.
- The run time environment will create an argument vector.
  - `argv` is the argument vector
  - `argc` is the number of arguments
- Argument vector is an array of pointers to strings.
- a *string* is an array of characters terminated by a binary 0 (NULL or '\0').
- *argv[0]* is always the program name, so *argc* is at least 1.

```
./try -g 2 fred
```



C Standard Header Files you may want to use

- **Standard Headers you should know about:**
  - `stdio.h` – file and console (also a file) IO: *`perror, printf, open, close, read, write, scanf`*, etc.
  - `stdlib.h` - common utility functions: *`malloc, calloc, strtol, atoi`*, etc
  - `string.h` - string and byte manipulation: *`strlen, strcpy, strcat, memcpy, memset`*, etc.
  - `ctype.h` – character types: *`isalnum, isprint, tolower,`* etc.
  - `errno.h` – defines *`errno`* used for reporting system errors
  - `math.h` – math functions: *`ceil, exp, floor, sqrt`*, etc.
  - `signal.h` – signal handling facility: *`raise, signal`*, etc
  - `time.h` – time related facility: *`asctime, clock, time_t,`* etc.

The Preprocessor

- **The C preprocessor permits you to define simple macros that are evaluated and expanded prior to compilation.**

- **Commands begin with a '#'. Abbreviated list:**
  - `#define` : defines a macro
  - `#include` : insert text from file
  - `#undef` : removes a macro definition
  - `#if` : conditional based on value of expression
  - `#ifdef` : conditional based on whether macro defined
  - `#else` : alternative
  - `#elif` : conditional alternative

Preprocessor: Macros

- **Using macros as functions, exercise caution:**
  - flawed example: `#define mymult(a,b) a*b`
    - Source: `k = mymult(i-1, j+5);`
    - Post preprocessing: `k = i - 1 * j + 5;`

Q. What's the correct form of macro?

use parenthesis positively.

Preprocessor: Conditional Compilation

- **`#ifdef __linux`**
  ```
  static inline int64_t
     gettime(void) {...}
  ```
- **`#elif defined(sun)`**
  ```
  static inline int64_t
     gettime(void) {return (int64_t)gethrtime()}
  ```
- **`#else`**
  ```
  static inline int64_t
     gettime(void) {... gettimeofday()...}
  ```
- **`#endif`**

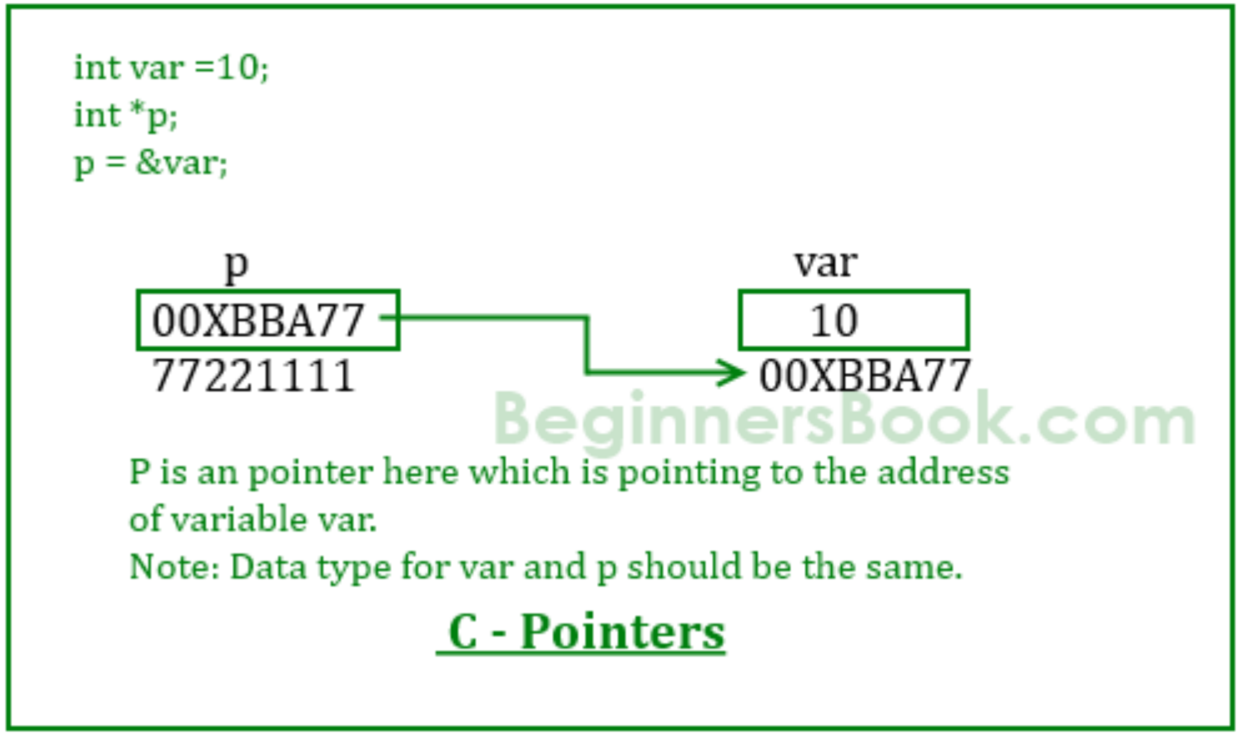
Another Simple C Program

```
int main (int argc, char **argv) {
  int i;
  printf("There are %d arguments\n", argc);
  for (i = 0; i < argc; i++)
    printf("Arg %d = %s\n", i, argv[i]);

  return 0;
}
```

- Notice that the syntax is similar to Java
- What's new in the above simple program?
  - Pointers will give you the most trouble

Pointers

- **What are pointers?**
  - A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location.



```
int var =10;
int *p;
p = &var;
```

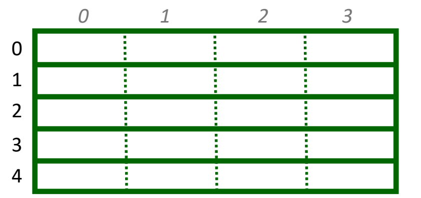
P is an pointer here which is pointing to the address of variable var.
Note: Data type for var and p should be the same.

C - Pointers

Arrays and Pointers

- **A variable declared as an array represents a contiguous region of memory in which the array elements are stored.**

```
int x[5]; // an array of 5 ints
```

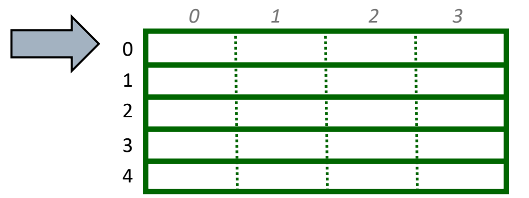


memory layout for array x

- **All arrays begin with an index of 0**
  - `x[0] = x[1] = x[2] = x[3] = x[4] = 1;`

- **An array identifier is equivalent to a pointer that references the first element of the array**
  - `int x[5], *ptr;`

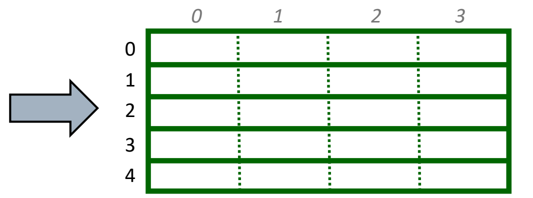
    `ptr = &x[0]` is equivalent to `ptr = x;`



memory layout for array x

- **Pointer arithmetic and arrays**
  - `int x[5];`
    - `x[2]` is the same as `*(x + 2)`,

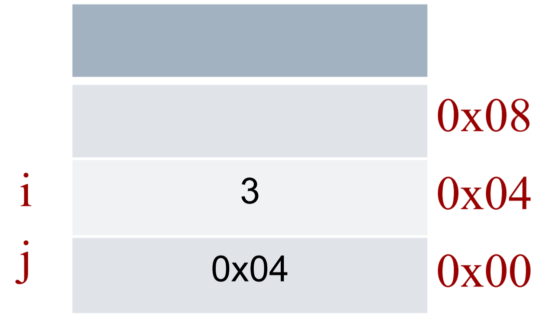
    the compiler will assume you mean 2 objects beyond element x.



memory layout for array x

Pointers

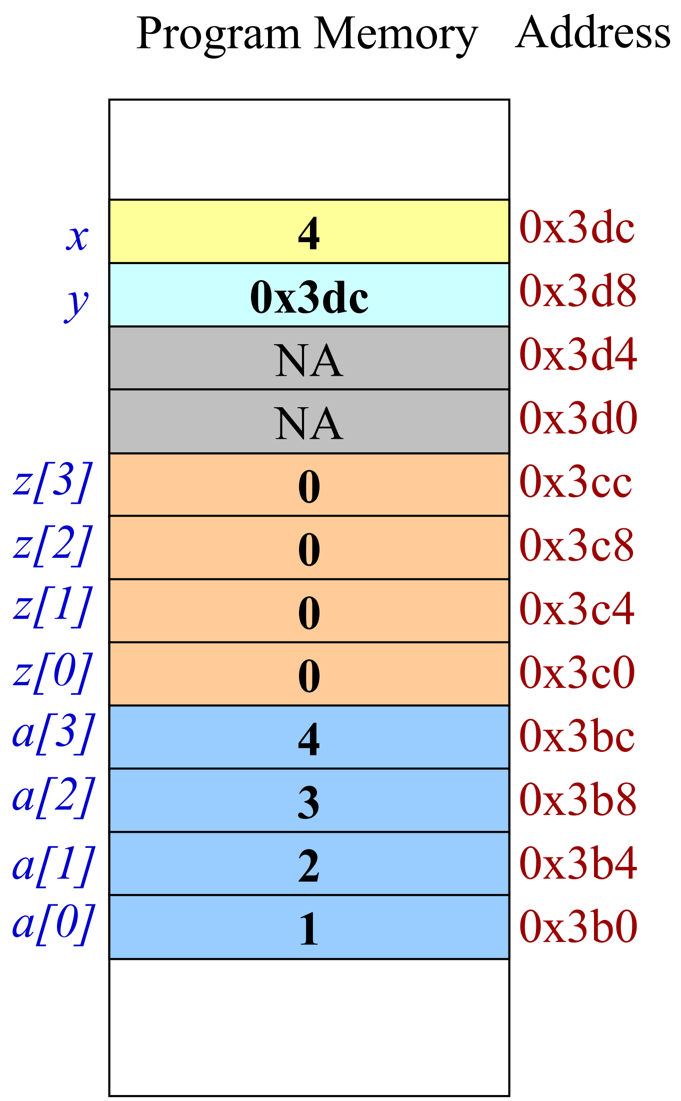
- **For any type T, you may form a pointer type to T.**
  - Pointers may reference a function or an object.
  - The value of a pointer is the address of the corresponding object or function
  - Examples: `int *i; char *x; int (*myfunc)();`
- **Pointer operators: * dereferences a pointer, & creates a pointer (reference to)**
  - `int i = 3;`
  - `int *j = &i;`
  - `*j = 4;`
  - `printf("i = %d\n", i); // prints i = 4`



- **Null pointers: use *NULL* or *0*. *It is a good idea to always initialize pointers to NULL.***

Pointers in C (and C++)

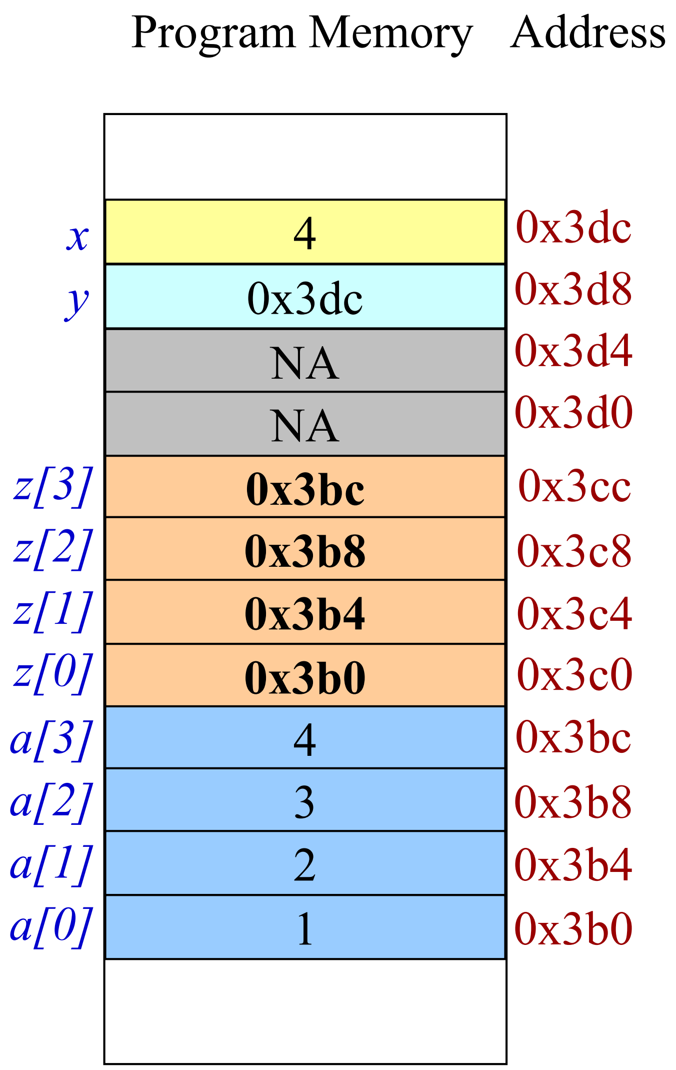
```
Step 1:
int main (int argc, argv) {
 int  x = 4;
 int *y = &x;
 int *z[4] = {NULL, NULL, NULL, NULL};
 int  a[4] = {1, 2, 3, 4};
...
```



Note: The compiler converts z[1] or *(z+1) to
Value at address (*Address of z* + *sizeof(int)*);

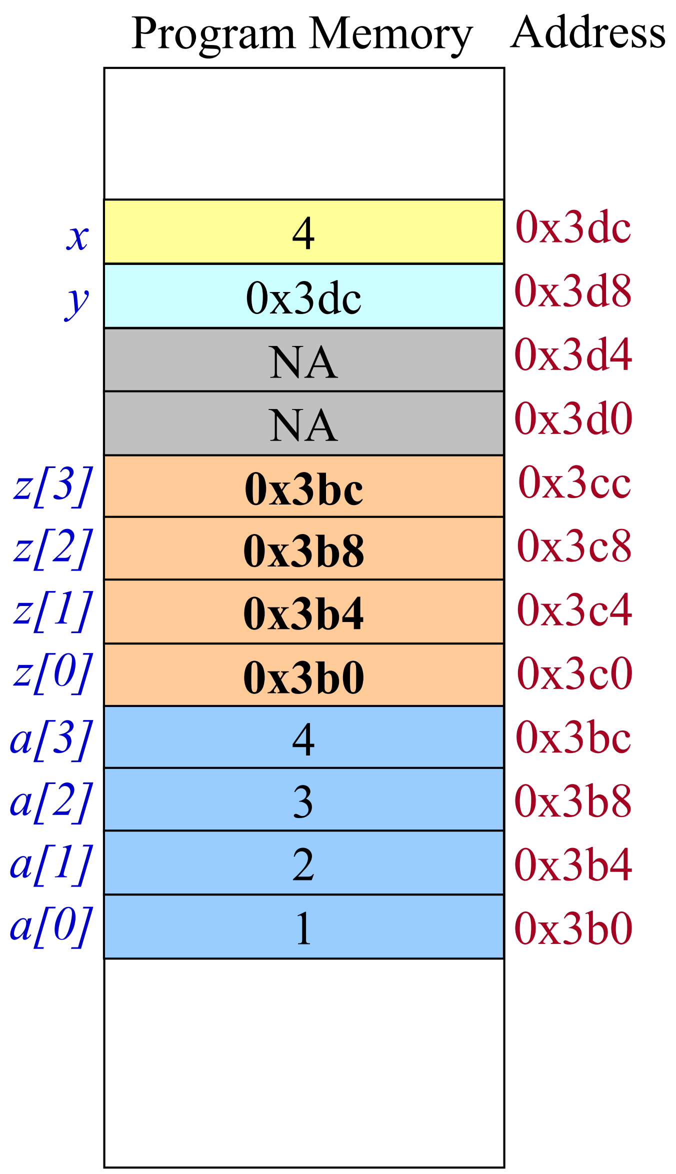
Pointers Continued

```
Step 1:
int main (int argc, argv) {
  int  x = 4;
  int *y = &x;
  int *z[4] = {NULL, NULL, NULL, NULL};
  int  a[4] = {1, 2, 3, 4};

Step 2: Assign addresses to array Z
  z[0] = a;     // same as &a[0];
  z[1] = a + 1; // same as &a[1];
  z[2] = a + 2; // same as &a[2];
  z[3] = a + 3; // same as &a[3];
```



Pointers Continued

```
Step 1:
int main (int argc, argv) {
 int x = 4;
 int *y = &x;
 int *z[4] = {NULL, NULL, NULL, NULL};
 int a[4] = {1, 2, 3, 4};

Step 2:
 z[0] = a;
 z[1] = a + 1;
 z[2] = a + 2;
 z[3] = a + 3;

Step 3: No change in z's values
 z[0] = (int *)((char *)a);
 z[1] = (int *)((char *)a
                + sizeof(int));
 z[2] = (int *)((char *)a
                + 2 * sizeof(int));
 z[3] = (int *)((char *)a
                + 3 * sizeof(int));
```



	Program Memory	Address
x	4	0x3dc
y	0x3dc	0x3d8
	NA	0x3d4
	NA	0x3d0
z[3]	**0x3bc**	0x3cc
z[2]	**0x3b8**	0x3c8
z[1]	**0x3b4**	0x3c4
z[0]	**0x3b0**	0x3c0
a[3]	4	0x3bc
a[2]	3	0x3b8
a[1]	2	0x3b4
a[0]	1	0x3b0

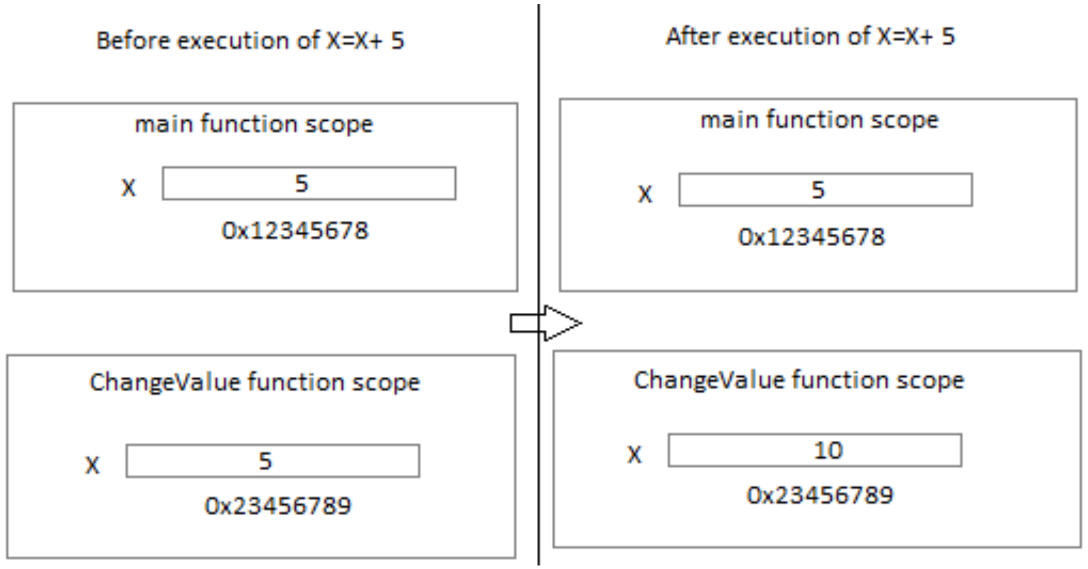
Contents

Functions

- **Always use function prototypes**

```
int myfunc (char *, int, struct MyStruct *);
int myfunc_noargs (void);
void myfunc_noreturn (int i);
```

- **C and C++ are *call by value*, copy of parameter passed to function**
  - if you want to alter the parameter then pass a pointer to it

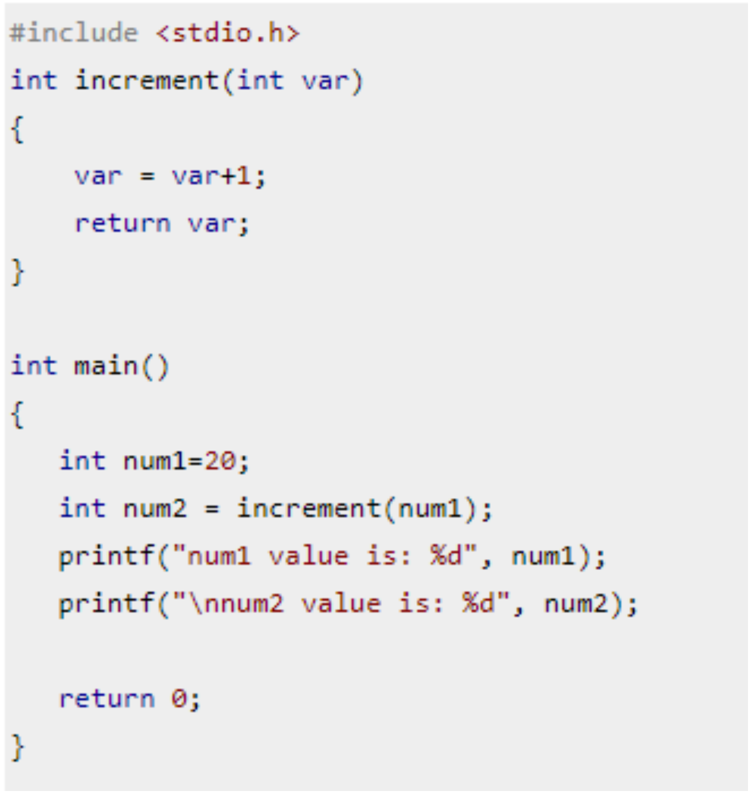


Before execution of X=X+ 5
main function scope
X
5
0x12345678
ChangeValue function scope
X
5
0x23456789
After execution of X=X+ 5
main function scope
X
5
0x12345678
ChangeValue function scope
X
10
0x23456789

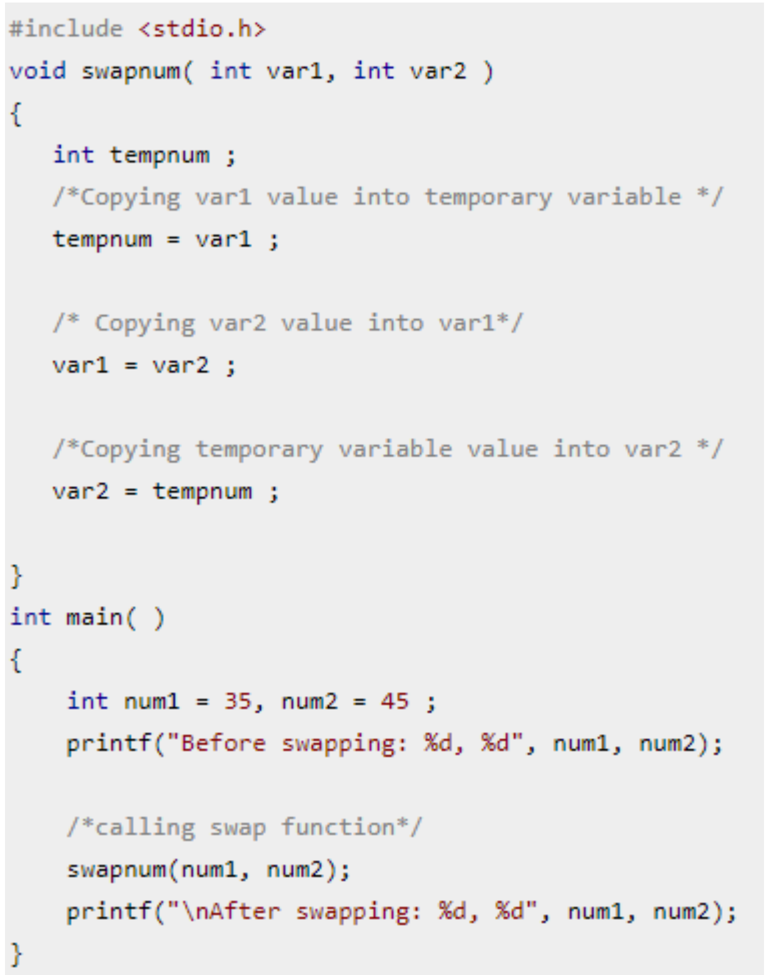
Call By Value vs. Call By Reference

Call by value

- The values of original parameters are copied to the variables in the function
- Operations performed on the formal parameters don’t reflect in the original parameters.



```
#include <stdio.h>
int increment(int var)
{
    var = var+1;
    return var;
}

int main()
{
   int num1=20;
   int num2 = increment(num1);
   printf("num1 value is: %d", num1);
   printf("\nnum2 value is: %d", num2);

   return 0;
}
```

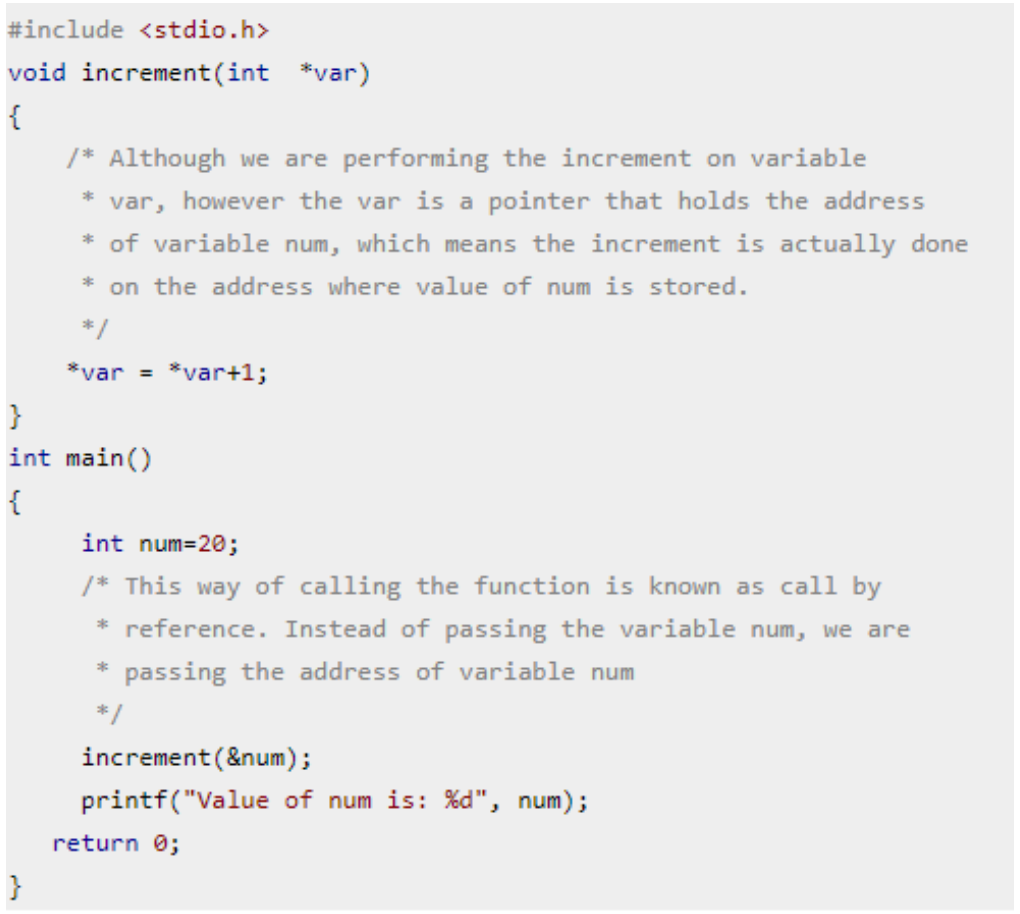


```
#include <stdio.h>
void swapnum( int var1, int var2 )
{
   int tempnum ;
   /*Copying var1 value into temporary variable */
   tempnum = var1 ;

   /* Copying var2 value into var1*/
   var1 = var2 ;

   /*Copying temporary variable value into var2 */
   var2 = tempnum ;

}
int main( )
{
    int num1 = 35, num2 = 45 ;
    printf("Before swapping: %d, %d", num1, num2);

    /*calling swap function*/
    swapnum(num1, num2);
    printf("\nAfter swapping: %d, %d", num1, num2);
}
```

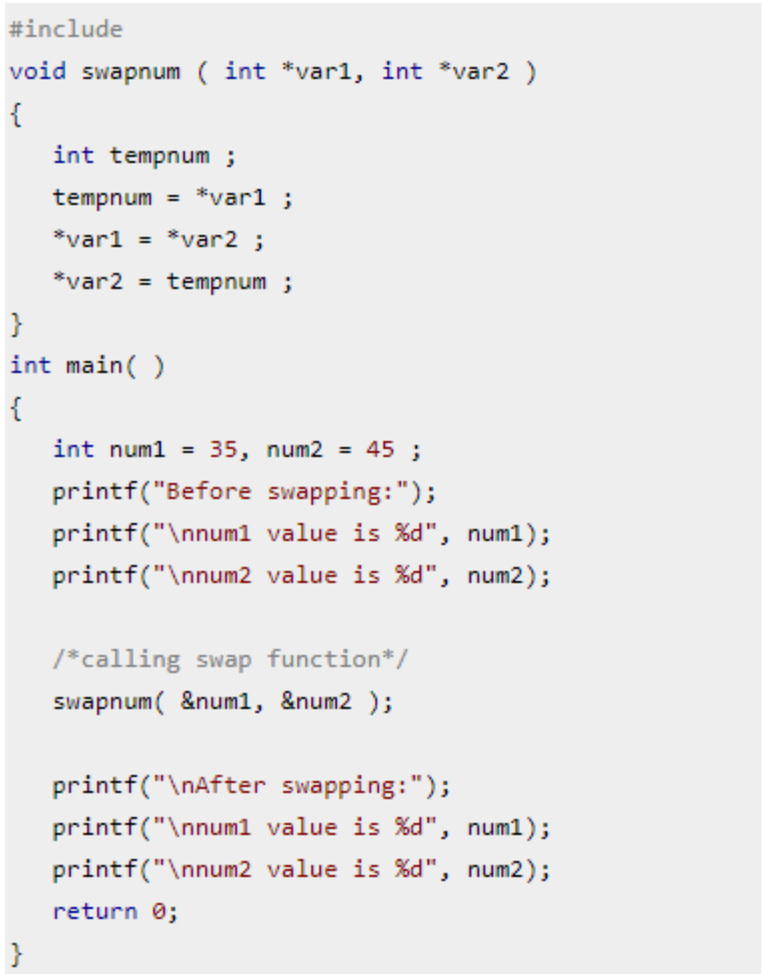
Call By Value vs. Call By Reference

Call by reference

- The operation performed on original parameters, affects the value of actual parameters because all the operations performed on the value stored in the address of actual parameters



```c
#include <stdio.h>
void increment(int  *var)
{
    /* Although we are performing the increment on variable
     * var, however the var is a pointer that holds the address
     * of variable num, which means the increment is actually done
     * on the address where value of num is stored.
     */
    *var = *var+1;
}
int main()
{
     int num=20;
     /* This way of calling the function is known as call by
      * reference. Instead of passing the variable num, we are
      * passing the address of variable num
      */
     increment(&num);
     printf("Value of num is: %d", num);
   return 0;
}
```



```c
#include
void swapnum ( int *var1, int *var2 )
{
   int tempnum ;
   tempnum = *var1 ;
   *var1 = *var2 ;
   *var2 = tempnum ;
}
int main( )
{
   int num1 = 35, num2 = 45 ;
   printf("Before swapping:");
   printf("\nnum1 value is %d", num1);
   printf("\nnum2 value is %d", num2);

   /*calling swap function*/
   swapnum( &num1, &num2 );

   printf("\nAfter swapping:");
   printf("\nnum1 value is %d", num1);
   printf("\nnum2 value is %d", num2);
   return 0;
}
```

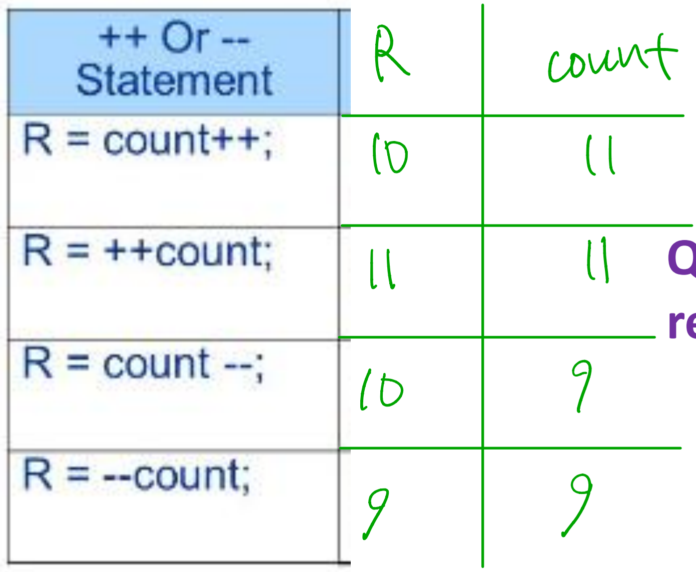
Basic Types

- **Basic data types**
  - Types: *char, int, float,* and *double*
  - Qualifiers: *short, long, unsigned, signed,* and *const*
- **Constant: 0x1234, 12, “Some string”**
- **Enumeration**
  - Names in different enumerations must be distinct

  -
    ```
    enum WeekDay_t {Mon, Tue, Wed, Thur, Fri};
    enum WeekendDay_t {Sat = 0, Sun = 4};
    ```

Operators

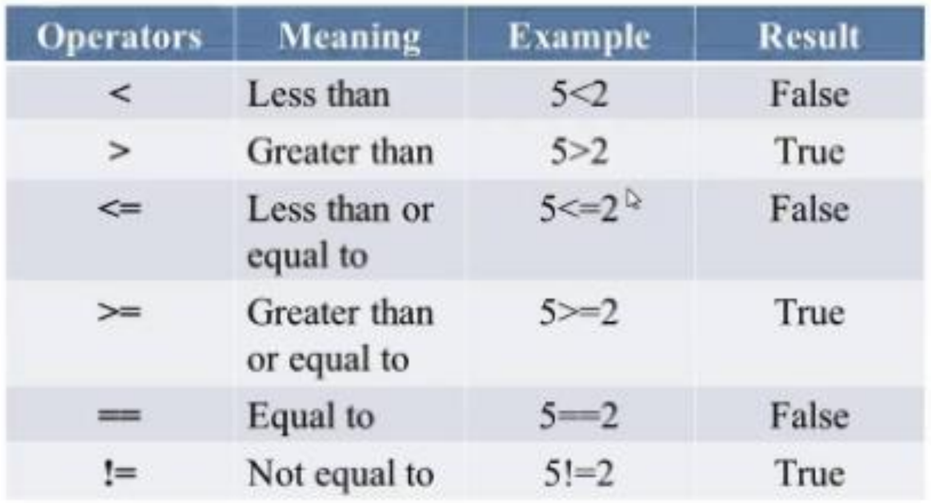
- **Arithmetic: +, -, *, /, %**
  - prefix ++i or --i ; increment/decrement before value is used
  - postfix i++, i--; increment/decrement after value is used



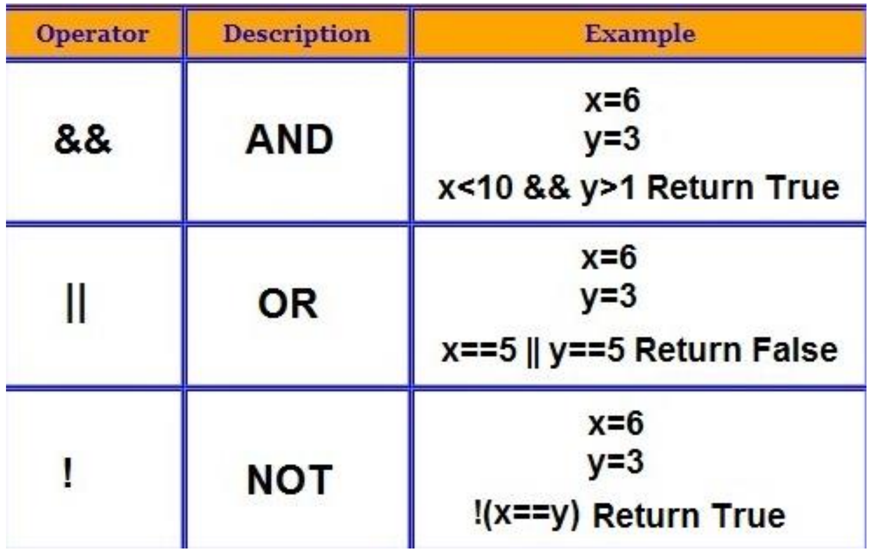
++ Or -- Statement	R	count
R = count++;	10	11
R = ++count;	11	11
R = count --;	10	9
R = --count;	9	9

Q. if count = 10, what's the result of R and count?

Relational and logical: <, >, <=, >=, ==, !=, &&, ||



Operators	Meaning	Example	Result
<	Less than	5<2	False
>	Greater than	5>2	True
<=	Less than or equal to	5<=2	False
>=	Greater than or equal to	5>=2	True
==	Equal to	5==2	False
!=	Not equal to	5!=2	True



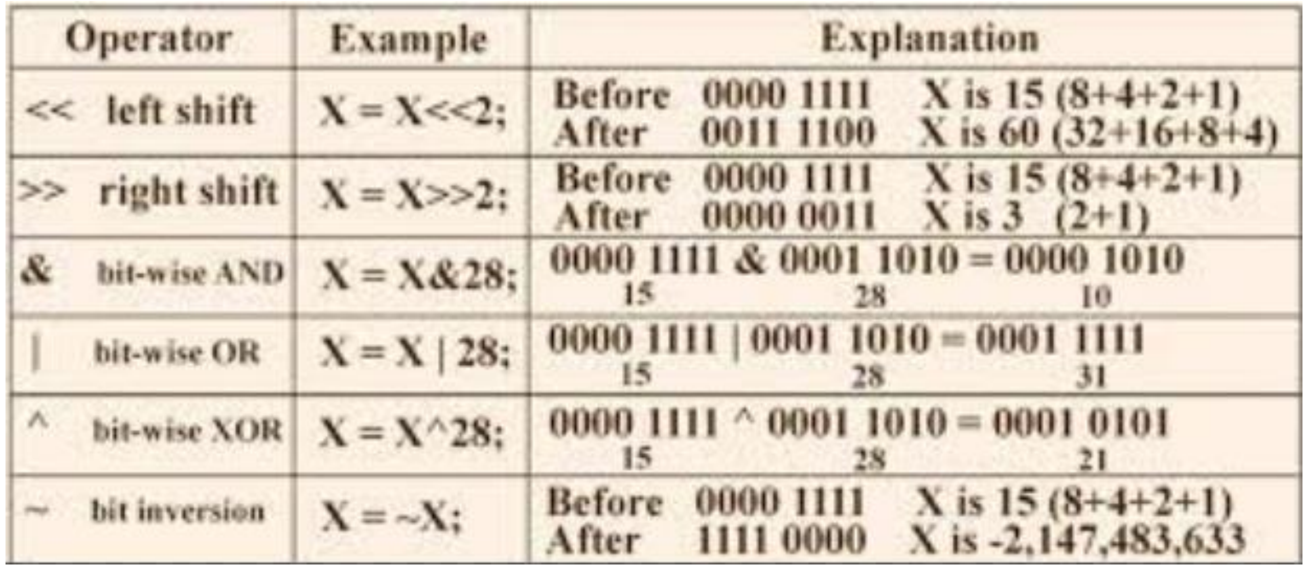
Operator	Description	Example

&&	AND	x=6
		y=3
		x<10 && y>1 Return True

||	OR	x=6
		y=3
		x==5 || y==5 Return False

!	NOT	x=6
		y=3
		!(x==y) Return True

- **Bitwise: &, |, ^ (xor), <<, >>, ~(ones complement)**



Operator		Example	Explanation
<<	left shift	X = X<<2;	Before 0000 1111 X is 15 (8+4+2+1) After 0011 1100 X is 60 (32+16+8+4)
>>	right shift	X = X>>2;	Before 0000 1111 X is 15 (8+4+2+1) After 0000 0011 X is 3 (2+1)
&	bit-wise AND	X = X&28;	0000 1111 & 0001 1010 = 0000 1010 15 28 10
|	bit-wise OR	X = X | 28;	0000 1111 | 0001 1010 = 0001 1111 15 28 31
^	bit-wise XOR	X = X^28;	0000 1111 ^ 0001 1010 = 0001 0101 15 28 21
~	bit inversion	X = ~X;	Before 0000 1111 X is 15 (8+4+2+1) After 1111 0000 X is -2,147,483,633

XOR → only 1 true ⇒ true!
OR → At least 1 true ⇒ true!
AND → At most 2 true ⇒ true!

Operator Precedence (from "C a Reference Manual", 5th Edition)

Tokens	Operator	Class	Precedence	Associates
names, *literals*	simple tokens	primary	16	n/a
a[k]	subscripting	postfix		left-to-right
f(...)	function call	postfix		left-to-right
.	direct selection	postfix		left-to-right
->	indirect selection	postfix		left to right
++ --	increment, decrement	**postfix**		left-to-right
(*type*){*init*}	compound literal	postfix		left-to-right
++ --	increment, decrement	**prefix**	15	right-to-left
sizeof	size	unary		right-to-left
~	bitwise not	unary		right-to-left
!	logical not	unary		right-to-left
- +	negation, plus	unary		right-to-left
&	address of	unary		right-to-left
*	indirection (*dereference*)	unary		right-to-left
(*type*)	casts	unary	14	right-to-left
* / %	multiplicative	binary	13	left-to-right
+ -	additive	binary	12	left-to-right
<< >>	left, right shift	binary	11	left-to-right
< <= > >=	relational	binary	10	left-to-right
== !=	equality/ineq.	binary	9	left-to-right
&	bitwise and	binary	8	left-to-right
^	bitwise xor	binary	7	left-to-right
|	bitwise or	binary	6	left-to-right
&&	logical and	binary	5	left-to-right
||	logical or	binary	4	left-to-right
?:	conditional	ternary	3	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	assignment	binary	2	right-to-left
,	sequential eval.	binary	1	left-to-right

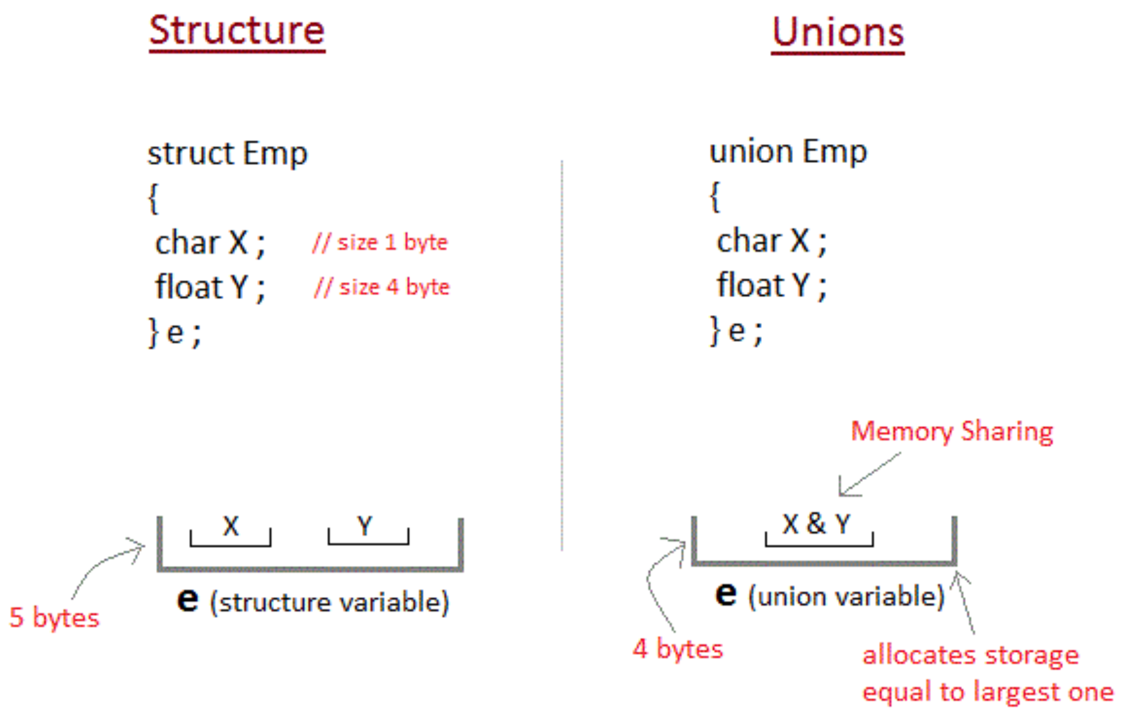
Structs and Unions

- **structures**

```
struct MyPoint {int x, int y};
typedef struct MyPoint MyPoint_t;
MyPoint_t point, *ptr;

point.x = 0;  point.y = 10;
ptr = &point; ptr->x = 12; ptr->y = 40;
```

unions

- Can only use one of the elements. Memory will be allocated for the largest element



Structure

```
struct Emp
{
 char X ;    // size 1 byte
 float Y ;   // size 4 byte
} e ;
```

Unions

```
union Emp
{
 char X ;
 float Y ;
} e ;
```

Q. Why unions are needed?

Save memory space

Conditional Statements (if/else)

```
if (a < 10)
  printf("a is less than 10\n");
else if (a == 10)
  printf("a is 10\n");
else
  printf("a is greater than 10\n");
```

- **If you have compound statements then use brackets (blocks)**

```
if (a < 4 && b > 10) {
  c = a * b; b = 0;
  printf("a = %d, a\'s address = 0x%08x\n", a, (uint32_t)&a);
} else {
  c = a + b; b = a;
}
```

Q. Is this correct?

```
if (a) x = 3;
else if (b) x = 2;
else (z) x = 0;
else x = -2;
```
→ Improper

Conditional Statements (switch)

```
int c = 10;
switch (c) {
  case 0:
    printf(“c is 0\n”);
    break;          mandatory
   case 1:
    printf(“c is 0\n”);
    break;
   ...
  default:
    printf(“Unknown value of c\n”);
    break;          not necessary
}
```

Q. What if we remove the break statement in each case? execute other case's statement also.

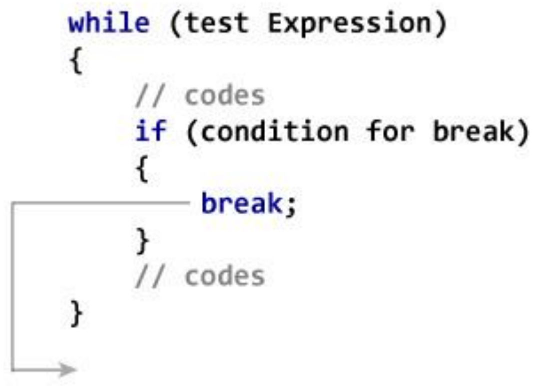
Q. Do we need the final break statement on the default case? Not mandatory.

Loops

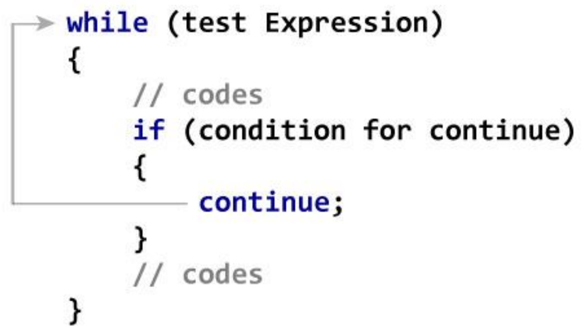
```
for (i = 0; i < MAXVALUE; i++) {
     dowork();
}


while (c != 12) {
     dowork();
}


do {
     dowork();
} while (c < 12);
```

flow control

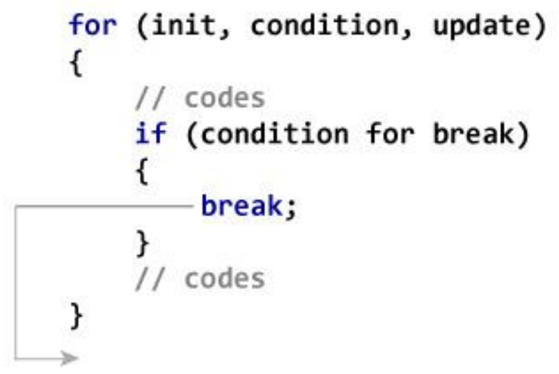
- **break** – exit innermost loop
- **continue** – perform next iteration of loop



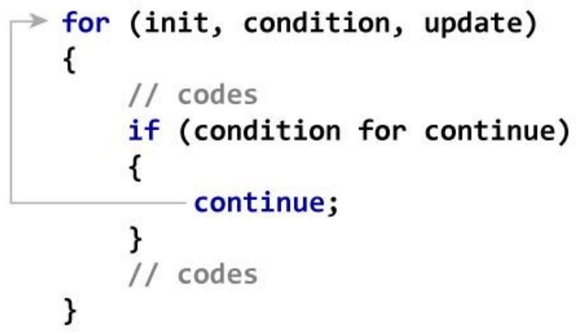
```
while (test Expression)
{
    // codes
    if (condition for break)
    {
        break;
    }
    // codes
}
```



```
while (test Expression)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```



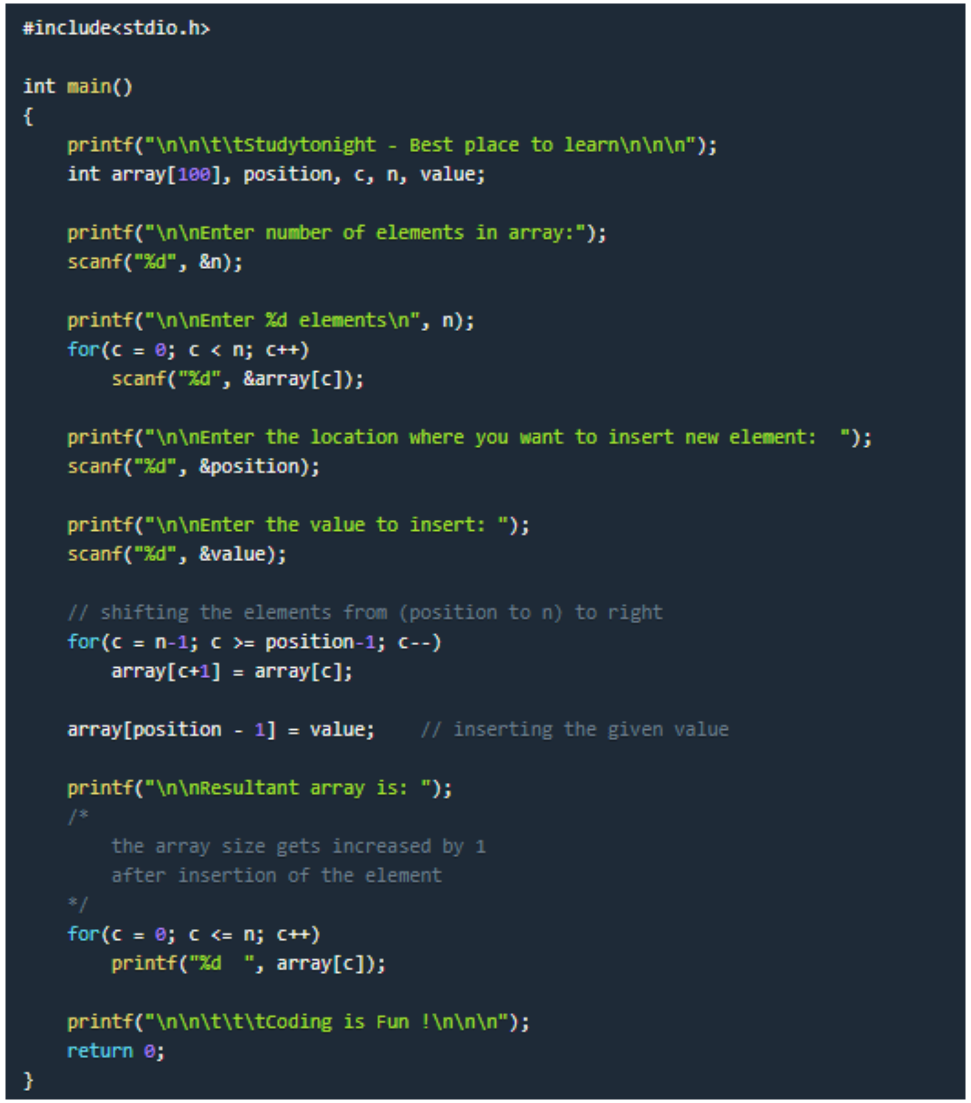
```
for (init, condition, update)
{
    // codes
    if (condition for break)
    {
        break;
    }
    // codes
}
```



```
for (init, condition, update)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```

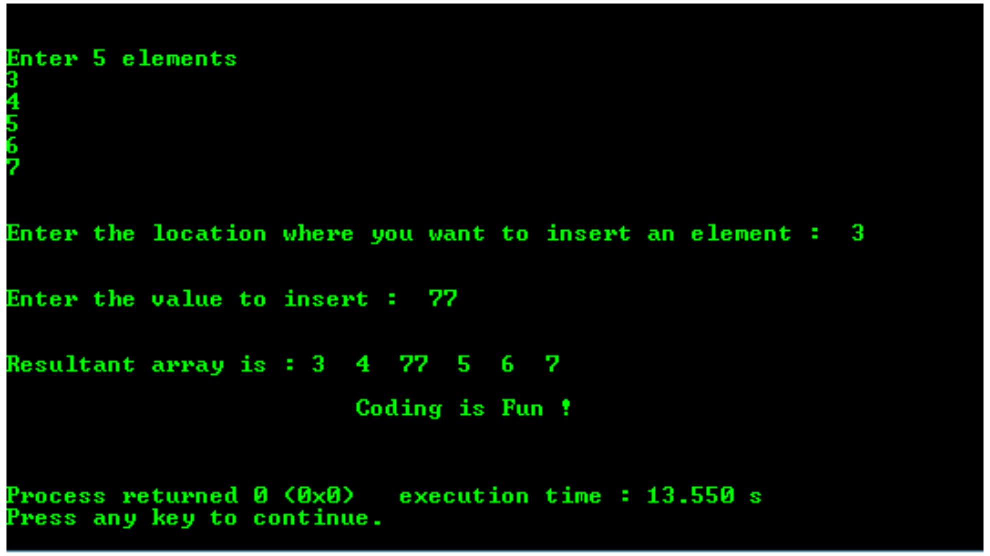
Practice2: Example #1

- **Program to insert an element in an Array**



```c
#include<stdio.h>

int main()
{
    printf("\n\n\t\tStudytonight - Best place to learn\n\n\n");
    int array[100], position, c, n, value;

    printf("\n\nEnter number of elements in array:");
    scanf("%d", &n);

    printf("\n\nEnter %d elements\n", n);
    for(c = 0; c < n; c++)
        scanf("%d", &array[c]);

    printf("\n\nEnter the location where you want to insert new element:  ");
    scanf("%d", &position);

    printf("\n\nEnter the value to insert: ");
    scanf("%d", &value);

    // shifting the elements from (position to n) to right
    for(c = n-1; c >= position-1; c--)
        array[c+1] = array[c];

    array[position - 1] = value;    // inserting the given value

    printf("\n\nResultant array is: ");
    /*
        the array size gets increased by 1
        after insertion of the element
    */
    for(c = 0; c <= n; c++)
        printf("%d  ", array[c]);

    printf("\n\n\t\t\tCoding is Fun !\n\n\n");
    return 0;
}
```

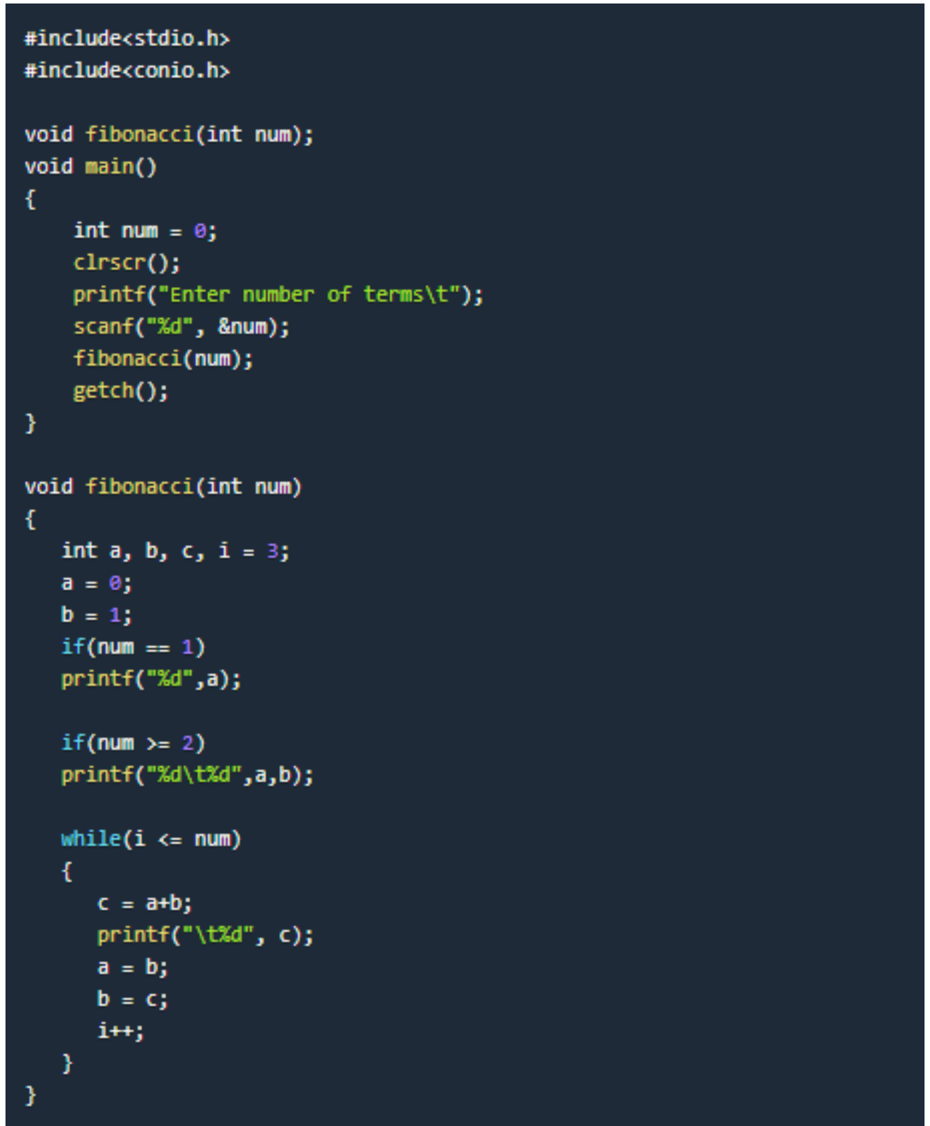
Output:



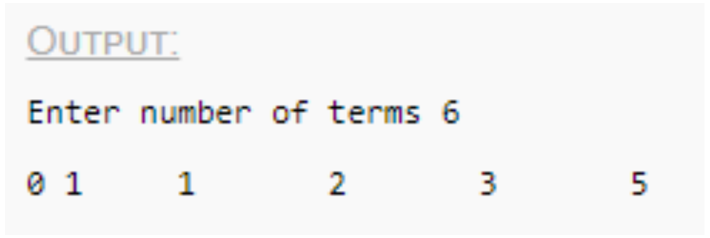
```
Enter 5 elements
3
4
5
6
7


Enter the location where you want to insert an element :  3

Enter the value to insert :  77

Resultant array is : 3  4  77  5  6  7
                    Coding is Fun !


Process returned 0 (0x0)   execution time : 13.550 s
Press any key to continue.
```

Practice2: Example #2

- **Program to print the Fibonacci Series**



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

void fibonacci(int num);
void main()
{
    int num = 0;
    clrscr();
    printf("Enter number of terms\t");
    scanf("%d", &num);
    fibonacci(num);
    getch();
}

void fibonacci(int num)
{
   int a, b, c, i = 3;
   a = 0;
   b = 1;
   if(num == 1)
   printf("%d",a);

   if(num >= 2)
   printf("%d\t%d",a,b);

   while(i <= num)
   {
      c = a+b;
      printf("\t%d", c);
      a = b;
      b = c;
      i++;
   }
}
```



OUTPUT:

```
Enter number of terms 6

0 1     1       2       3       5
```

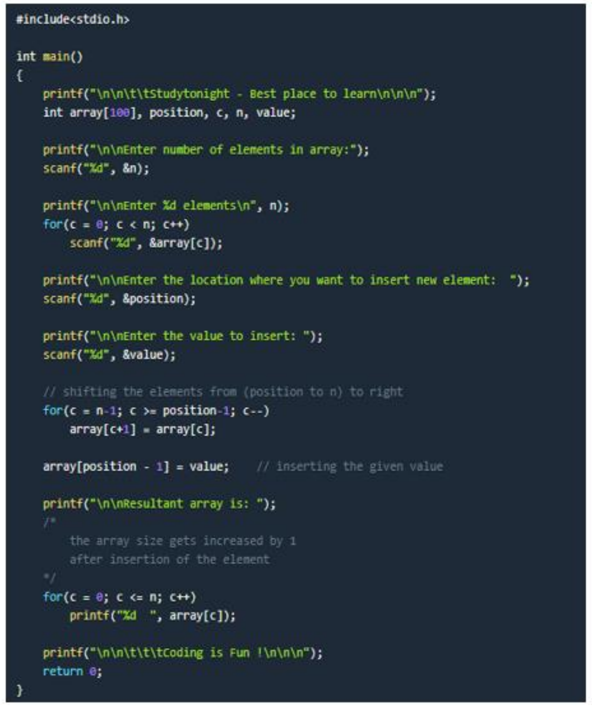
Homework Assignment #1

1. Modify Example #1

- Make 10 elements
- Insert **YOUR STUDENT ID** in 5-th elements
- **Submission**: capture the output to show **YOUR STUDENT ID**

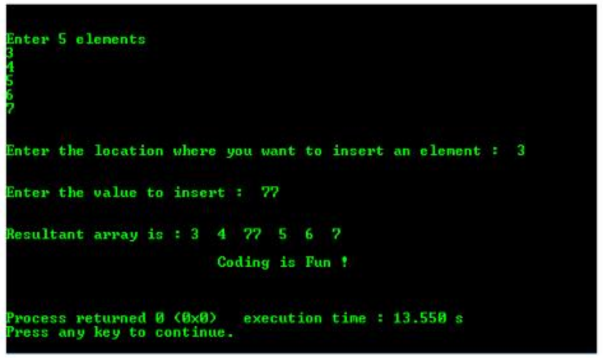
Example #1

Program to insert an element in an Array



```c
#include<stdio.h>

int main()
{
    printf("\n\n\t\tStudytonight - Best place to learn\n\n\n");
    int array[100], position, c, n, value;

    printf("\n\nEnter number of elements in array:");
    scanf("%d", &n);

    printf("\n\nEnter %d elements\n", n);
    for(c = 0; c < n; c++)
        scanf("%d", &array[c]);

    printf("\n\nEnter the location where you want to insert new element:  ");
    scanf("%d", &position);

    printf("\n\nEnter the value to insert: ");
    scanf("%d", &value);

    // shifting the elements from (position to n) to right
    for(c = n-1; c >= position-1; c--)
        array[c+1] = array[c];

    array[position - 1] = value;    // inserting the given value

    printf("\n\nResultant array is: ");
    /*
        the array size gets increased by 1
        after insertion of the element
    */
    for(c = 0; c <= n; c++)
        printf("%d  ", array[c]);

    printf("\n\n\t\t\tCoding is Fun !\n\n\n");
    return 0;
}
```

Output:

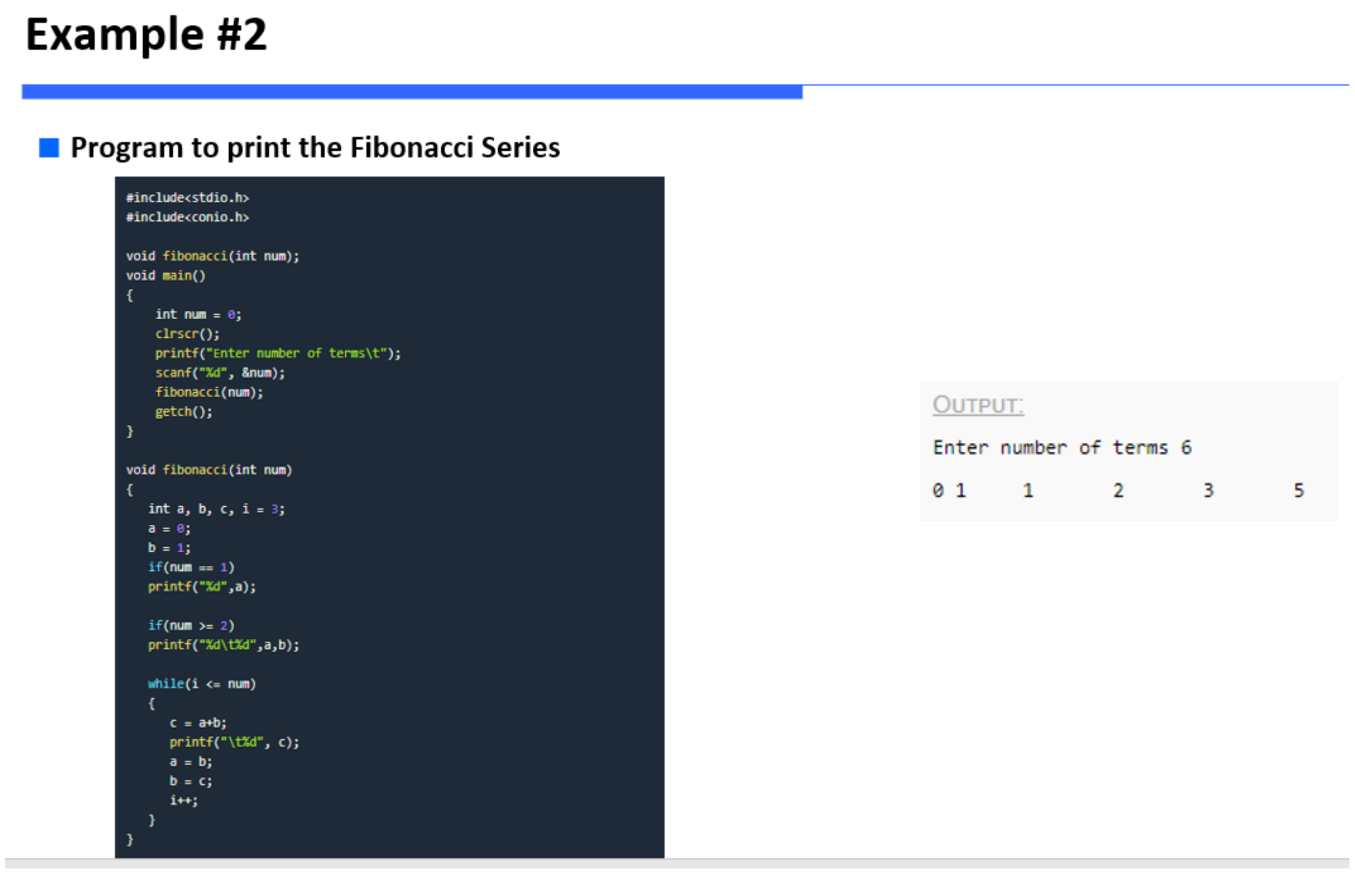


```
Enter 5 elements
3
4
5
6
7


Enter the location where you want to insert an element :  3


Enter the value to insert :  77


Resultant array is : 3  4  77  5  6  7
                        Coding is Fun !



Process returned 0 (0x0)   execution time : 13.550 s
Press any key to continue.
```

Homework Assignment #1

2. Modify Example #2

- Change while-loop into for-loop
- **Submission**: capture the **source code**



Example #2

■ **Program to print the Fibonacci Series**

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

void fibonacci(int num);
void main()
{
    int num = 0;
    clrscr();
    printf("Enter number of terms\t");
    scanf("%d", &num);
    fibonacci(num);
    getch();
}

void fibonacci(int num)
{
   int a, b, c, i = 3;
   a = 0;
   b = 1;
   if(num == 1)
   printf("%d",a);

   if(num >= 2)
   printf("%d\t%d",a,b);

   while(i <= num)
   {
      c = a+b;
      printf("\t%d", c);
      a = b;
      b = c;
      i++;
   }
}
```

OUTPUT:

```
Enter number of terms 6
0 1     1       2       3       5
```

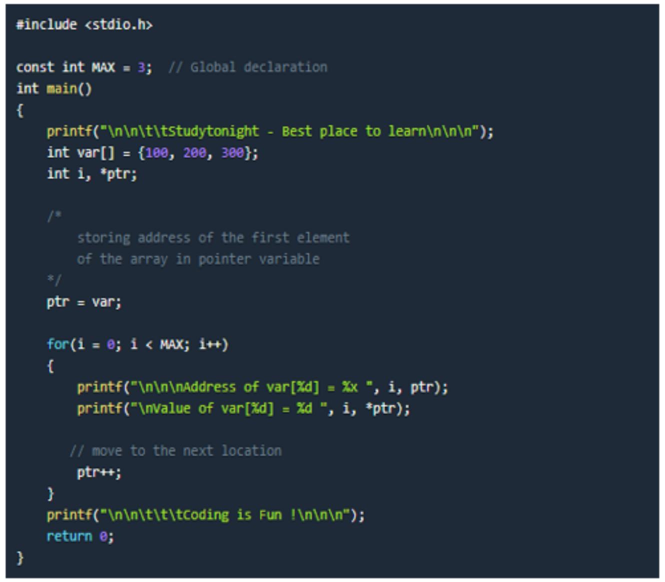
Homework Assignment #1

3. Modify Example #3

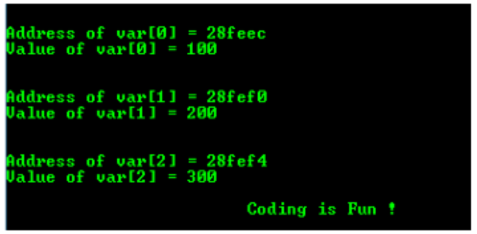
- Add one more element in the array whose value is **YOUR STUDENT ID**
- Print four elements in the array including **YOUR STUDENT ID**
- **Submission**: capture the output to show **YOUR STUDENT ID**

Example #3

- **Accessing array elements by incrementing a Pointer**



```c
#include <stdio.h>

const int MAX = 3;  // Global declaration
int main()
{
    printf("\n\n\t\tStudytonight - Best place to learn\n\n\n");
    int var[] = {100, 200, 300};
    int i, *ptr;

    /*
        storing address of the first element
        of the array in pointer variable
    */
    ptr = var;

    for(i = 0; i < MAX; i++)
    {
        printf("\n\n\nAddress of var[%d] = %x ", i, ptr);
        printf("\nValue of var[%d] = %d ", i, *ptr);

       // move to the next location
        ptr++;
    }
    printf("\n\n\t\t\tCoding is Fun !\n\n\n");
    return 0;
}
```



```
Address of var[0] = 28feec
Value of var[0] = 100

Address of var[1] = 28fef0
Value of var[1] = 200

Address of var[2] = 28fef4
Value of var[2] = 300
                        Coding is Fun !
```

The End