Computer Programming II

Ming-Feng Tsai (Victor Tsai)

Dept. of Computer Science
National Chengchi University

C Revisited

`switch` and `break`

- Example: calculator.c

```
23          switch (operator) {
24              case '+':
25                  result += value;
26                  break;
27              case '-':
28                  result -= value;
29                  break;
30              case '*':
31                  result *= value;
32                  break;
33              case '/':
34                  if (value == 0) {
35                      printf("Error:Divide by zero\n");
36                      printf("   operation ignored\n");
37                  } else
38                      result /= value;
39                  break;
40              default:
41                  printf("Unknown operator %c\n", operator);
42                  break;
43          }
```

`switch` and `break`

- Example: calculator.c

```
23          switch (operator) {
24              case '+':
25                  result += value;
26                  break;
27              case '-':
28                  result -= value;
29                  break;
30              case '*':
31                  result *= value;
32                  break;
33              case '/':
34                  if (value == 0) {
35                      printf("Error:Divide by zero\n");
36                      printf("   operation ignored\n");
37                  } else
38                      result /= value;
39                  break;
40              default:
41                  printf("Unknown operator %c\n", operator);
42                  break;
43          }
```

switch, **break**, and **continue**

```
#include <stdio.h>

int   number;       /* Number we are converting */
char  type;         /* Type of conversion to do */
char  line[80];     /* input line */

int main(void)
{
    while (1) {

        printf("Enter conversion and number: ");

        fgets(line, sizeof(line), stdin);
        sscanf(line, "%c", &type);

        if ((type == 'q') || (type == 'Q'))
            break;

        switch (type) {
            case 'o':
            case 'O':           /* Octal conversion */
                sscanf(line, "%c %o", &type, &number);
                break;
            case 'x':
            case 'X':           /* Hexadecimal conversion */
                sscanf(line, "%c %x", &type, &number);
                break;
            case 'd':
            case 'D':           /* Decimal (For completeness) */
                sscanf(line, "%c %d", &type, &number);
                break;
            case '?':
            case 'h':           /* Help */
                printf("Letter  Conversion\n");
                printf("  o     Octal\n");
                printf("  x     Hexadecimal\n");
                printf("  d     Decimal\n");
                printf("  q     Quit program\n");

                /* Don't print the number */
                continue;
            default:
                printf("Type ? for help\n");
                /* Don't print the number */
                continue;
        }
        printf("Result is %d\n", number);
    }
    return (0);
}
```

break (leave the switch)

continue (within the while loop)

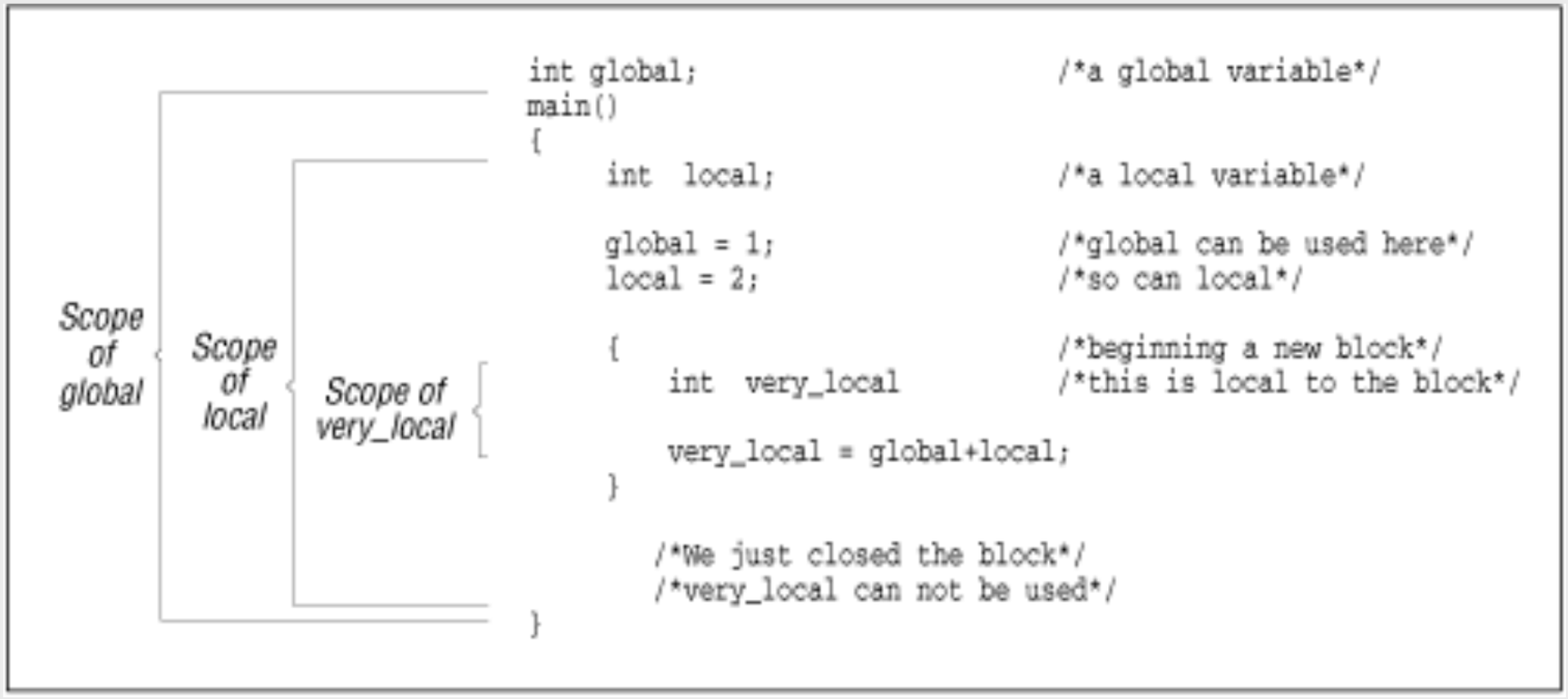
break (leave the while loop)

More about Variables and Functions

Scope and Class

- All variables have two attributes
  - scope
    - global vs. local variables
  - class
    - permanent vs. temporary

Local and Global Variables




```
int global;                     /*a global variable*/
main()
{
     int  local;                /*a local variable*/

     global = 1;                /*global can be used here*/
     local = 2;                 /*so can local*/

     {                          /*beginning a new block*/
         int  very_local        /*this is local to the block*/

         very_local = global+local;
     }

        /*We just closed the block*/
        /*very_local can not be used*/
}
```

Hidden Variables

Scope of global variable count.

Local variable count *hides global variable* count *in this area.*

```
int total;                          /*total number of entries*/
int count;                          /*count of total entries*/

main()
{

     total = 0;
     count = 0;                       /*set global counter*/


     {
          int  count;                 /*a local counter*/

          count=0;

          while (1) {
             if (count > 10)
                 break;

             total += count;
             ++count;
          }
     {

     ++count;
     return (0);
}
```

Hidden Variables

Scope of global variable count.

Local variable count *hides global variable* count *in this area.*

```
int total;                          /*total number of entries*/
int count;                          /*count of total entries*/

main()
{

     total = 0;
     count = 0;                       /*set global counter*/


     {
          int  count;                 /*a local counter*/

          count=0;

          while (1) {
             if (count > 10)
                 break;

             total += count;
             ++count;
          }
     {

     ++count;
     return (0);
}
```

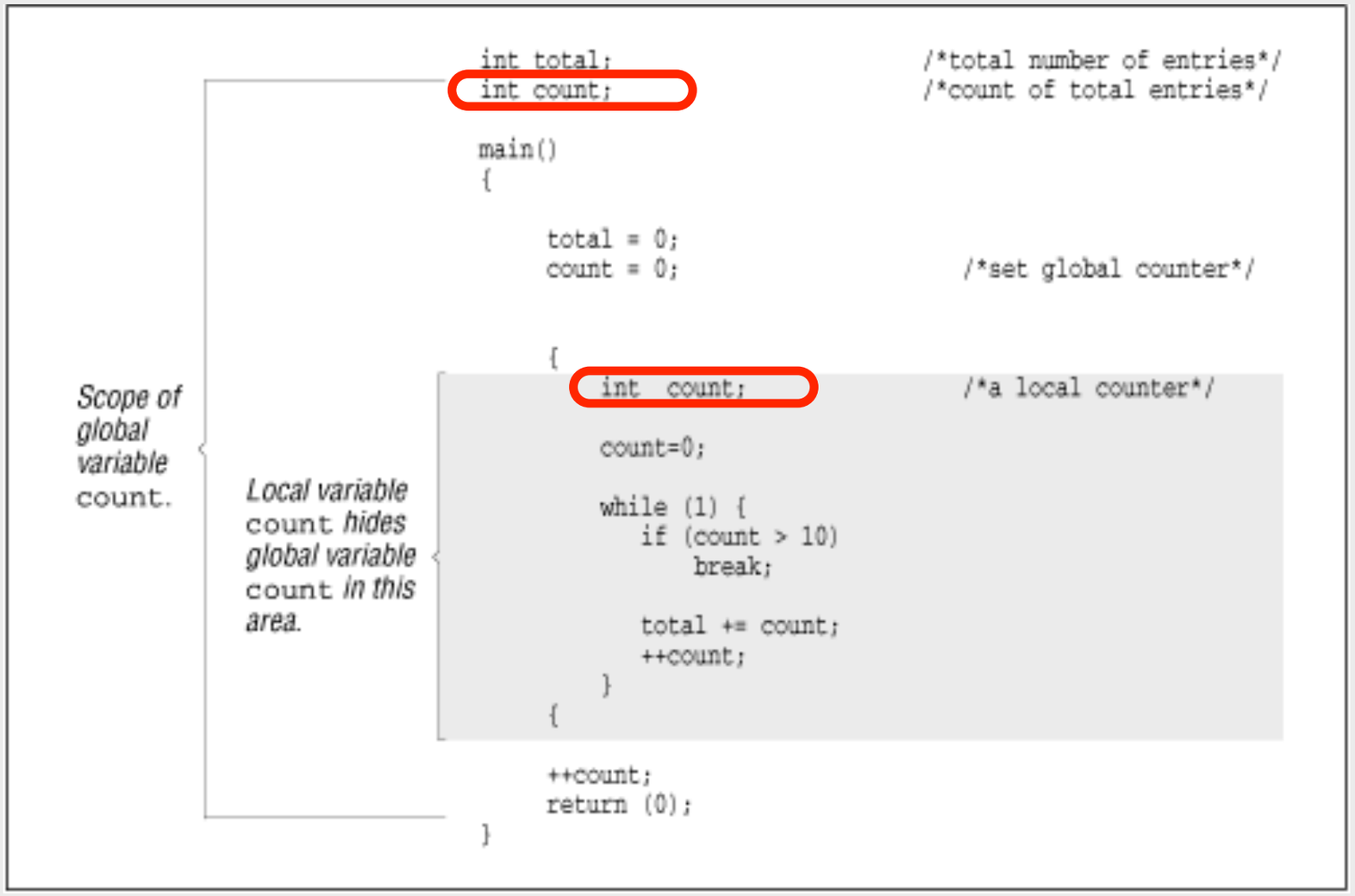
Hidden Variables

Scope of global variable count.

Local variable count *hides global variable* count *in this area.*

```
int total;                          /*total number of entries*/
int count;                          /*count of total entries*/

main()
{

     total = 0;
     count = 0;                        /*set global counter*/


     {
        int  count;                    /*a local counter*/

        count=0;

        while (1) {
           if (count > 10)
               break;

           total += count;
           ++count;
        }
     {

     ++count;
     return (0);
}
```

Hidden Variables



The global count has bee hidden by the local count for the scope of the block

The Class of Variables

- The class of variable
  - **permanent**: created and initialized before the program starts and remain in until it terminates
    - **temporary**: allocated from stack; may cause “Stack overflow” error if try to allocate too many temporary variables
      - The size of the stack depends on the system and compiler you are using

static

static

- Local variables are temporary unless they are declared **static**

static

- Local variables are temporary unless they are declared **static**
- When using with global variables, **static** indicates that a variable is local to the current file

permanent vs. temporary

- Example: vars.c

```
4     int counter;     /* loop counter */
5     for (counter = 0; counter < 3; ++counter) {
6         int temporary = 1; /* A temporary variable */
7         static int permanent = 1; /* A permanent variable */
8
9         printf("Temporary %d Permanent %d\n",
10            temporary, permanent);
11        ++temporary;
12        ++permanent;
13    }
14    return (0);
```

permanent vs. temporary

- Example: vars.c

```
4     int counter;     /* loop counter */
5     for (counter = 0; counter < 3; ++counter) {
6         int temporary = 1; /* A temporary variable */
7         static int permanent = 1; /* A permanent variable */
8
9         printf("Temporary %d Permanent %d\n",
10            temporary, permanent);
11        ++temporary;
12        ++permanent;
13    }
14    return (0);
```

permanent vs. temporary

- Example: vars.c

```
4     int counter;    /* loop counter */
5     for (counter = 0; counter < 3; ++counter) {
6         int temporary = 1; /* A temporary variable */
7         static int permanent = 1; /* A permanent variable */
8
9         printf("Temporary %d Permanent %d\n",
10            temporary, permanent);
11        ++temporary;
12        ++permanent;
13    }
14    return (0);
```

```
Temporary 1 Permanent 1
Temporary 1 Permanent 2
Temporary 1 Permanent 3
```

Declaration Modifiers

Declared	Scope	Class	Initialized
Outside all blocks	Global	Permanent	Once
static outside all blocks	Global[1]	Permanent	Once
Inside a block	Local	Temporary	Each time block is entered
static inside a block	Local	Permanent	Once

Declaration Modifiers

Declared	Scope	Class	Initialized
Outside all blocks	Global	Permanent	Once
static outside all blocks	Global[1]	Permanent	Once
Inside a block	Local	Temporary	Each time block is entered
static inside a block	Local	Permanent	Once

A static declaration made outside blocks indicates the variable is local to the file in which it is declared

Functions

- Each function contains the following
  - Name
  - Description
  - Parameters
  - Returns

Call by Value

Call by Value

- When a function is called the parameters are copied -- “call by value”

Call by Value

- When a function is called the parameters are copied -- “call by value”
- The function is unable to change any variable passed as a parameter

Call by Value

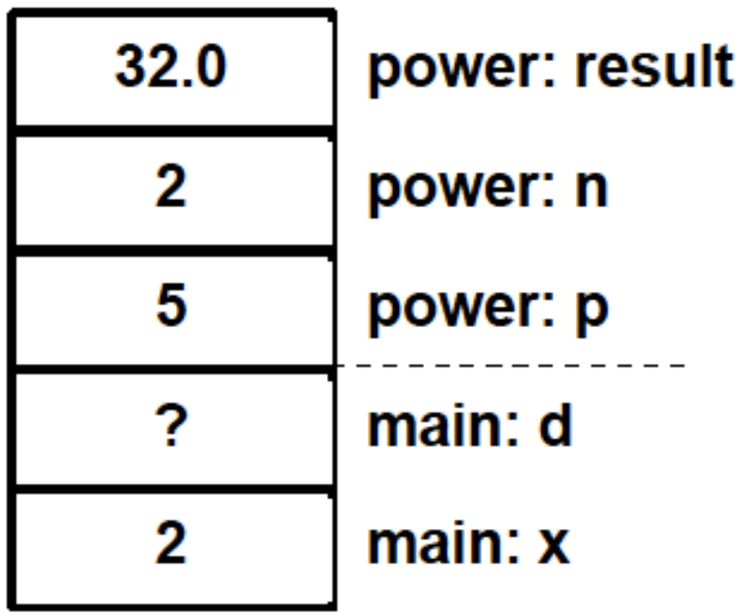
- When a function is called the parameters are copied -- “call by value”
- The function is unable to change any variable passed as a parameter
- In the chapter of pointers, “call by reference” will be discussed

C and the Stack

- C uses a stack to store local variables (i.e. those declared in functions), it is also used when passing parameters to functions
  - The calling function pushes the parameters
  - The function is called
  - The called function picks up the parameters
  - The called function pushes its local variables
  - When finished, the called function pops its local variables and jumps back to the calling function
  - The calling function pops the parameters
  - The return value is handled

Stack Example

```
#include <stdio.h>

double power(int, int);

int  main(void)
{
     int      x = 2;
     double   d;

     d = power(x, 5);
     printf("%lf\n", d);

     return 0;
}

double power(int n, int p)
{
     double   result = n;

     while(--p > 0)
              result *= n;

     return result;
}
```



Storage

- C stores local variables on the stack
- Global variables may be declared. These are not stack based, but are placed in the data segment
- Special keywords exist to specify where local variables are stored:
  - **auto** - place on the stack (default)
  - **static** - place in the data segment (heap)
  - **register** - place in a CPU register
- Data may also be placed on the heap, this will be discussed in a later chapter

auto

- Local variables are automatically allocated on entry into, and automatically deallocated on exit from, a function
- These variables are therefore called “automatic”
  - Initial value: random
  - Initialization: recommended

auto

- Local variables are automatically allocated on entry into, and automatically deallocated on exit from, a function
- These variables are therefore called “automatic”
- Initial value: random
- Initialization: recommended

```
int  table(void)
{
     int        lines = 13;
     auto int   columns;
```

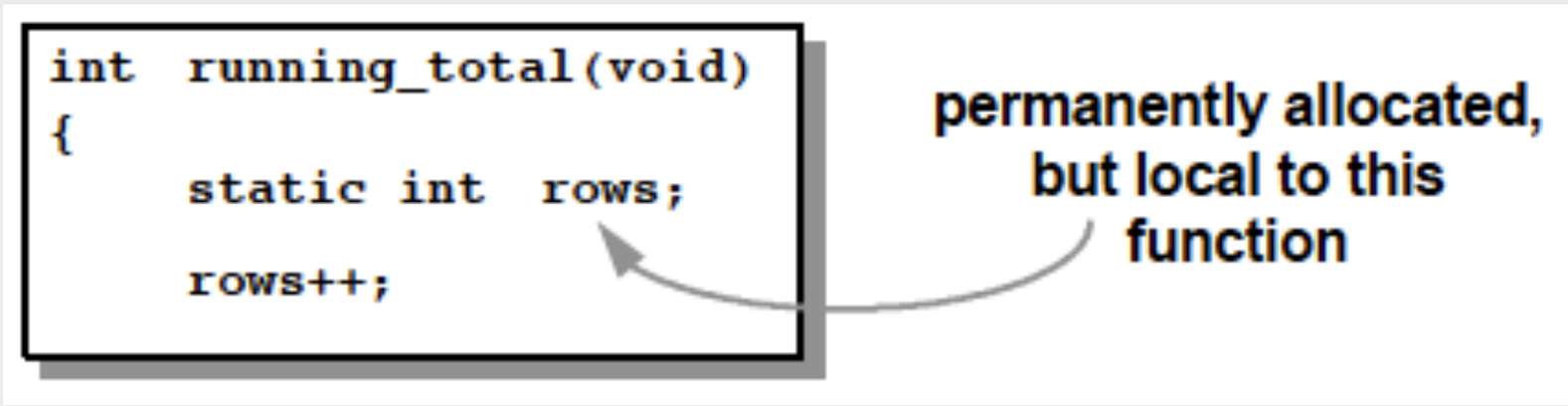
auto keyword redundant

`static`

- The static keyword instructs the compiler to place a variable into the data segment
- The data segment is permanent (static)
- A value left in a static in one call to a function will still be there at the next call
  - Initial value: 0
  - Initialization: unnecessary if you like zeros

static

- The static keyword instructs the compiler to place a variable into the data segment
- The data segment is permanent (static)
- A value left in a static in one call to a function will still be there at the next call
- Initial value: 0
- Initialization: unnecessary if you like zeros



static

- Example: static.c

```
5  void f1() {
6      static int k = 0; /*static variable has its local scope*/
7      int j = 10; /*local variable*/
8      printf("value of k = %d, j = %d", k, j);
9      k += 10;
10     j += 10;
11 }
```

```
13 void f2() {
14     static int k = 0; /*static variable has its local scope*/
15     int j = 10; /*local variable*/
16     printf("value of k = %d, j = %d", k, j);
17     k += 10;
18     j += 10;
19 }
```

```
value of k = 0, j = 10 after first call of f1
value of k = 10, j = 10 after second call of f1
value of k = 20, j = 10 after third call of f1
value of k = 0, j = 10 after first call of f2
value of k = 10, j = 10 after second call of f2
value of k = 20, j = 10 after third call of f2
```

static

- Example: static.c

```
5  void f1() {
6      static int k = 0; /*static variable has its local scope*/
7      int j = 10; /*local variable*/
8      printf("value of k = %d, j = %d", k, j);
9      k += 10;
10     j += 10;
11 }
```

```
13 void f2() {
14     static int k = 0; /*static variable has its local scope*/
15     int j = 10; /*local variable*/
16     printf("value of k = %d, j = %d", k, j);
17     k += 10;
18     j += 10;
19 }
```

```
value of k = 0, j = 10 after first call of f1
value of k = 10, j = 10 after second call of f1
value of k = 20, j = 10 after third call of f1
value of k = 0, j = 10 after first call of f2
value of k = 10, j = 10 after second call of f2
value of k = 20, j = 10 after third call of f2
```

static

- Example: static.c

```
5  void f1() {
6      static int k = 0; /*static variable has its local scope*/
7      int j = 10; /*local variable*/
8      printf("value of k = %d, j = %d", k, j);
9      k += 10;
10     j += 10;
11 }
```

```
13 void f2() {
14     static int k = 0; /*static variable has its local scope*/
15     int j = 10; /*local variable*/
16     printf("value of k = %d, j = %d", k, j);
17     k += 10;
18     j += 10;
19 }
```

```
value of k = 0, j = 10 after first call of f1
value of k = 10, j = 10 after second call of f1
value of k = 20, j = 10 after third call of f1
value of k = 0, j = 10 after first call of f2
value of k = 10, j = 10 after second call of f2
value of k = 20, j = 10 after third call of f2
```

static

- Example: static.c

```c
void f1() {
    static int k = 0; /*static variable has its local scope*/
    int j = 10; /*local variable*/
    printf("value of k = %d, j = %d", k, j);
    k += 10;
    j += 10;
}
```

```c
void f2() {
    static int k = 0; /*static variable has its local scope*/
    int j = 10; /*local variable*/
    printf("value of k = %d, j = %d", k, j);
    k += 10;
    j += 10;
}
```

```
value of k = 0, j = 10 after first call of f1
value of k = 10, j = 10 after second call of f1
value of k = 20, j = 10 after third call of f1
value of k = 0, j = 10 after first call of f2
value of k = 10, j = 10 after second call of f2
value of k = 20, j = 10 after third call of f2
```

static

- Example: static.c

```
5  void f1() {
6      static int k = 0; /*static variable has its local scope*/
7      int j = 10; /*local variable*/
8      printf("value of k = %d, j = %d", k, j);
9      k += 10;
10     j += 10;
11 }
```

```
13 void f2() {
14     static int k = 0; /*static variable has its local scope*/
15     int j = 10; /*local variable*/
16     printf("value of k = %d, j = %d", k, j);
17     k += 10;
18     j += 10;
19 }
```

```
value of k = 0, j = 10 after first call of f1
value of k = 10, j = 10 after second call of f1
value of k = 20, j = 10 after third call of f1
value of k = 0, j = 10 after first call of f2
value of k = 10, j = 10 after second call of f2
value of k = 20, j = 10 after third call of f2
```

static

- Example: static.c

```
5  void f1() {
6      static int k = 0; /*static variable has its local scope*/
7      int j = 10; /*local variable*/
8      printf("value of k = %d, j = %d", k, j);
9      k += 10;
10     j += 10;
11 }
```

```
13 void f2() {
14     static int k = 0; /*static variable has its local scope*/
15     int j = 10; /*local variable*/
16     printf("value of k = %d, j = %d", k, j);
17     k += 10;
18     j += 10;
19 }
```

static variables have local scope!!

```
value of k = 0, j = 10 after first call of f1
value of k = 10, j = 10 after second call of f1
value of k = 20, j = 10 after third call of f1
value of k = 0, j = 10 after first call of f2
value of k = 10, j = 10 after second call of f2
value of k = 20, j = 10 after third call of f2
```

`register`

- The register keyword tells the compiler to place a variable into a CPU register for optimization purpose
- If a register is unavailable the request will be ignored
- Largely redundant with optimizing compilers
  - Initial value: random
  - Initialization: recommended

register

- The register keyword tells the compiler to place a variable into a CPU register for optimization purpose
- If a register is unavailable the request will be ignored
- Largely redundant with optimizing compilers
- Initial value: random
- Initialization: recommended

```
void  speedy_function(void)
{
      register int i;

      for(i = 0; i < 10000; i++)
```

Global Variables

- Global variables are created by placing the declaration outside all functions
- They are placed in the data segment
- Initial value: 0
- Initialization: unnecessary if you like zeros

Global Variables

- Global variables are created by placing the declaration outside all functions
- They are placed in the data segment
- Initial value: 0
- Initialization: unnecessary if you like zeros

```
#include <stdio.h>

double    d;

int main(void)
{
     int   i;

     return 0;
}
```

variable “d” is global and available to all functions defined below it

Local Variable

- Example: length.c

```
3  int length(char string[]) {
4      int index; /* index into the string */
5
6      /*
7       * Loop until we reach the end of string character
8       */
9      for (index = 0; string[index] != '\0'; ++index)
10         /* do nothing */
11     return (index);
12 }
```

```
14 int main() {
15     char line[100];»/* Input line from user */
16
17     while (1) {
18         printf("Enter line:");
19         fgets(line, sizeof(line), stdin);
20
21         printf("Length (including newline) is: %d\n", length(line)-1);
22     }
23 }
```

Local Variable

- Example: length.c

```c
int length(char string[]) {
    int index; /* index into the string */

    /*
     * Loop until we reach the end of string character
     */
    for (index = 0; string[index] != '\0'; ++index)
        /* do nothing */
    return (index);
}
```

```
Enter line:Hello
Length (including newline) is: -1
Enter line:World!!
Length (including newline) is: -1
Enter line:Hello
Length (including newline) is: -1
Enter line:^C
```

```c
int main() {
    char line[100];»/* Input line from user */

    while (1) {
        printf("Enter line:");
        fgets(line, sizeof(line), stdin);

        printf("Length (including newline) is: %d\n", length(line)-1);
    }
}
```

Local Variable

- Example: length.c

```c
int length(char string[]) {
    int index; /* index into the string */

    /*
     * Loop until we reach the end of string character
     */
    for (index = 0; string[index] != '\0'; ++index)
        /* do nothing */
    return (index);
}
```

```
Enter line:Hello
Length (including newline) is: -1
Enter line:World!!
Length (including newline) is: -1
Enter line:Hello
Length (including newline) is: -1
Enter line:^C
```

?

```c
int main() {
    char line[100];»/* Input line from user */

    while (1) {
        printf("Enter line:");
        fgets(line, sizeof(line), stdin);

        printf("Length (including newline) is: %d\n", length(line)-1);
    }
}
```

Local Variable

- Example: length.c

```c
int length(char string[]) {
    int index; /* index into the string */

    /*
     * Loop until we reach the end of string character
     */
    for (index = 0; string[index] != '\0'; ++index)
        /* do nothing */
    return (index);
}
```

```
Enter line:Hello
Length (including newline) is: -1
Enter line:World!!
Length (including newline) is: -1
Enter line:Hello
Length (including newline) is: -1
Enter line:^C
```

?

```c
int main() {
    char line[100];»/* Input line from user */

    while (1) {
        printf("Enter line:");
        fgets(line, sizeof(line), stdin);

        printf("Length (including newline) is: %d\n", length(line)-1);
    }
}
```

Structured Programming

- Top-down programming
  - start at the top (main) and work your way down
- Bottom-up programming
  - write the lowest-level function first, testing it, and then building on that working set
- Both techniques are useful

Structured Programming

- Top-down programming
  - start at the top (main) and work your way down
- Bottom-up programming
  - write the lowest-level function first, testing it, and then building on that working set
- Both techniques are useful

```
int main()
{
    init();
    solve_problems();
    finish_up();
    return (0);
}
```

Recursion

Recursion

- Recursion

Recursion

- Recursion
  - A function calls itself directly or indirectly

Recursion

- Recursion
  - A function calls itself directly or indirectly
- A recursive function must follow two basic rules

Recursion

- Recursion
  - A function calls itself directly or indirectly
- A recursive function must follow two basic rules
  - It must have an ending point

Recursion

- Recursion
  - A function calls itself directly or indirectly
- A recursive function must follow two basic rules
  - It must have an ending point
  - It must make the problem simpler

Recursion

```
int fact(int number)
{
    if (number == 0)
        return (1);
    /* else */
    return (number * fact(number-1));
}
```

Recursion

```
int fact(int number)
{
    if (number == 0)
        return (1);
    /* else */
    return (number * fact(number-1));
}
```

5! = fact(5)
= 5 * fact(4)
= 4 * fact(3)
= 3 * fact(2)
= 2 * fact(1)
= 1 * fact(0)

Recursion

```
int fact(int number)
{
    if (number == 0)
        return (1);
    /* else */
    return (number * fact(number-1));
}
```

5! = fact(5)
= 5 * fact(4)
= 4 * fact(3)
= 3 * fact(2)
= 2 * fact(1)
= 1 * fact(0)

Recursion

```
int sum(int first, int last, int array[])
{
    if (first == last)
        return (array[first]);
    /* else */
    return (array[first] + sum(first+1, last, array));
}
```

Recursion

```
int sum(int first, int last, int array[])
{
    if (first == last)
        return (array[first]);
    /* else */
    return (array[first] + sum(first+1, last, array));
}
```

sum(1 8 3 2)
= 1 + sum(8 3 2)
= 8 + sum(3 2)
= 3 + sum(2)
= 2

Recursion

```
int sum(int first, int last, int array[])
{
    if (first == last)
        return (array[first]);
    /* else */
    return (array[first] + sum(first+1, last, array));
}
```

sum(1 8 3 2)
= 1 + sum(8 3 2)
= 8 + sum(3 2)
= 3 + sum(2)
= 2

Towers of Hanoi

Towers of Hanoi

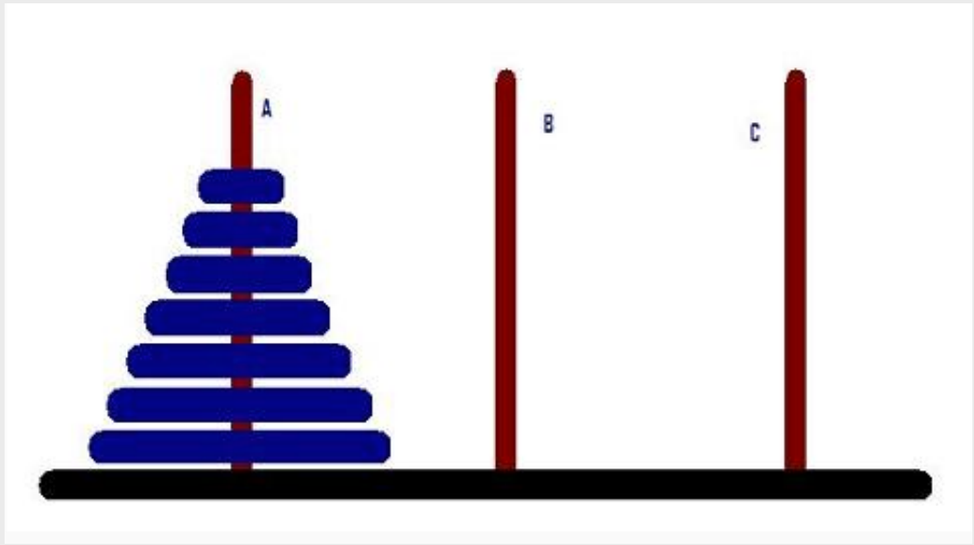
- The puzzle was invented by the French mathematician Edouard Lucas in 1883.

Towers of Hanoi

- The puzzle was invented by the French mathematician Edouard Lucas in 1883.
- Objective:transfer the entire tower to one of the other pegs, **moving only one disk at a time and never a larger one onto a smaller**

Towers of Hanoi

- The puzzle was invented by the French mathematician Edouard Lucas in 1883.
- Objective:transfer the entire tower to one of the other pegs, **moving only one disk at a time and never a larger one onto a smaller**



Towers of Hanoi

- Number of Switches

Towers of Hanoi

- Number of Switches

$$T_n = \begin{cases} 0 & \text{if} \quad n = 0 \\ 2T_{n-1} + 1 & \text{if} \quad n > 0 \end{cases}$$

Towers of Hanoi

- Number of Switches

$$T_n = \begin{cases} 0 & \text{if} \quad n = 0 \\ 2T_{n-1} + 1 & \text{if} \quad n > 0 \end{cases}$$

$$\text{let } S_n = T_n + 1$$

Towers of Hanoi

- Number of Switches

$$T_n = \begin{cases} 0 & \text{if} \quad n = 0 \\ 2T_{n-1} + 1 & \text{if} \quad n > 0 \end{cases}$$

$$\text{let } S_n = T_n + 1$$

$$\begin{aligned} S_n &= T_n + 1 \\ &= (2T_{n-1} + 1) + 1 \\ &= 2T_{n-1} + 2 \\ &= 2(T_{n-1} + 1) \\ &= 2S_{n-1} \end{aligned}$$

Towers of Hanoi

- Number of Switches

$$T_n = \begin{cases} 0 & \text{if} \quad n = 0 \\ 2T_{n-1} + 1 & \text{if} \quad n > 0 \end{cases}$$

$$\text{let } S_n = T_n + 1$$

$$\begin{aligned} S_n &= T_n + 1 \\ &= (2T_{n-1} + 1) + 1 \\ &= 2T_{n-1} + 2 \\ &= 2(T_{n-1} + 1) \\ &= 2S_{n-1} \end{aligned}$$

$$S_n = 2^n$$

$$\Longrightarrow T_n = S_n - 1 = 2^n - 1$$

Towers of Hanoi

- Example: tower.c
  - Animation1, Animation2

```c
void hanoi(int num_towers, char from, char tmp, char to) {
    if(num_towers == 1) {
        printf("Move sheet from %c to %c\n", from, to);
        num_switch++;
    }
    else {
        hanoi(num_towers-1, from, to, tmp);
        hanoi(1, from, tmp, to);
        hanoi(num_towers-1, tmp, from, to);
    }
}
```

Towers of Hanoi

- Example: tower.c
  - Animation1, Animation2

```
5  void hanoi(int num_towers, char from, char tmp, char to) {
6      if(num_towers == 1) {
7          printf("Move sheet from %c to %c\n", from, to);
8          num_switch++;
9      }
10     else {
11         hanoi(num_towers-1, from, to, tmp);
12         hanoi(1, from, tmp, to);
13         hanoi(num_towers-1, tmp, from, to);
14     }
15 }
```

Towers of Hanoi

- Example: tower.c
  - Animation1, Animation2

```
5  void hanoi(int num_towers, char from, char tmp, char to) {
6      if(num_towers == 1) {
7          printf("Move sheet from %c to %c\n", from, to);
8          num_switch++;
9      }
10     else {
11         hanoi(num_towers-1, from, to, tmp);
12         hanoi(1, from, tmp, to);
13         hanoi(num_towers-1, tmp, from, to);
14     }
15 }
```

C Preprocessors

C Preprocessor

C Preprocessor

- In the early days, when C was still being developed, it soon became apparent that C needed a facility for

C Preprocessor

- In the early days, when C was still being developed, it soon became apparent that C needed a facility for
  - handling named constants

C Preprocessor

- In the early days, when C was still being developed, it soon became apparent that C needed a facility for
  - handling named constants
  - macros

C Preprocessor

- In the early days, when C was still being developed, it soon became apparent that C needed a facility for
  - handling named constants
  - macros
  - include files

`#define` Statement

`#define` Statement

- All preprocessor commands begin with a hash mark (#) in column one

`#define` Statement

- All preprocessor commands begin with a hash mark (#) in column one
- Putting a semicolon at the end of a preprocessor directive can lead to unexpected results

`#define` Statement

- All preprocessor commands begin with a hash mark (#) in column one
- Putting a semicolon at the end of a preprocessor directive can lead to unexpected results
- It is common programming practice to use all uppercase letters for macro names

```
#define SIZE 20
```

`#define` Example

- Example: init2b.c

```c
#define SIZE 20     /* work on 20 elements */

int data[SIZE];     /* some data */
int twice[SIZE];    /* twice some data */

int main()
{
    int index;   /* index into the data */

    for (index = 0; index < SIZE; ++index) {
        data[index] = index;
        twice[index] = index * 2;
    }
    return (0);
}
```

`#define` Example

- Example: init2b.c

```c
#define SIZE 20     /* work on 20 elements */

int data[SIZE];     /* some data */
int twice[SIZE];    /* twice some data */

int main()
{
    int index;   /* index into the data */

    for (index = 0; index < SIZE; ++index) {
        data[index] = index;
        twice[index] = index * 2;
    }
    return (0);
}
```

`#define` Example

- Example: init2b.c

```c
#define SIZE 20     /* work on 20 elements */

int data[SIZE];     /* some data */
int twice[SIZE];    /* twice some data */

int main()
{
    int index;   /* index into the data */

    for (index = 0; index < SIZE; ++index) {
        data[index] = index;
        twice[index] = index * 2;
    }
    return (0);
}
```

`#define` Statement

`#define` Statement

- The general form of a simple define statement is:

```
#define name substitute-tex
```

`#define` Statement

- The general form of a simple define statement is:

```
#define name substitute-tex
```

- Example

`#define` Statement

- The general form of a simple define statement is:

  `#define name substitute-tex`

- Example

```
#define FOR_ALL for(i = 0; i < ARRAY_SIZE; i++)
```

and use it like this:

```
/*
 * Clear the array
 */
FOR_ALL {
    data[i] = 0;
}
```

`#define` Statement

- The general form of a simple define statement is:

  `#define name substitute-tex`

- Example

```
#define FOR_ALL for(i = 0; i < ARRAY_SIZE; i++)
```

and use it like this:

```
/*
 * Clear the array
 */
FOR_ALL {
    data[i] = 0;
}
```

`#define` Statement

- The general form of a simple define statement is:

  `#define name substitute-tex`

- Example

```
#define FOR_ALL for(i = 0; i < ARRAY_SIZE; i++)
```

and use it like this:

```
/*
 * Clear the array
 */
FOR_ALL {
    data[i] = 0;
}
```

`#define` Statement

- The general form of a simple define statement is:

  `#define name substitute-tex`

- Example

```
#define FOR_ALL for(i = 0; i < ARRAY_SIZE; i++)
```

and use it like this:

```
/*
 * Clear the array
 */
FOR_ALL {
    data[i] = 0;
}
```

```
#define BEGIN {
#define END }
. . .
    if (index == 0)
    BEGIN
        printf("Starting\n");
    END
```

`#define` Statement

- The general form of a simple define statement is:

  ```
  #define name substitute-tex
  ```

- Example

```
#define FOR_ALL for(i = 0; i < ARRAY_SIZE; i++)
```

and use it like this:

```
/*
 * Clear the array
 */
FOR_ALL {
    data[i] = 0;
}
```

```
#define BEGIN {
#define END }
. . .
    if (index == 0)
    BEGIN
        printf("Starting\n");
    END
```

`#define` Statement

- The general form of a simple define statement is:

  ```
  #define name substitute-tex
  ```

- Example

```
#define FOR_ALL for(i = 0; i < ARRAY_SIZE; i++)
```

and use it like this:

```
/*
 * Clear the array
 */
FOR_ALL {
    data[i] = 0;
}
```

```
#define BEGIN {
#define END }
. . .
    if (index == 0)
    BEGIN
        printf("Starting\n");
    END
```

`#define` Statement

- The general form of a simple define statement is:

  ```
  #define name substitute-tex
  ```

- Example

```
#define FOR_ALL for(i = 0; i < ARRAY_SIZE; i++)
```

and use it like this:

```
/*
 * Clear the array
 */
FOR_ALL {
    data[i] = 0;
}
```

```
#define BEGIN {
#define END }
. . .
    if (index == 0)
    BEGIN
        printf("Starting\n");
    END
```

`#define` Statement

`#define` Statement

```c
#define BIG_NUMBER 10 ** 10

main()
{
    /* index for our calculations */
    int   index;

    index = 0;

    /* syntax error on next line */
    while (index < BIG_NUMBER) {
        index = index * 8;
    }

    return (0);
}
```

`#define` Statement

- The example cannot be compiled!! Why??

```
1  #define BIG_NUMBER 10 ** 10
2
3  main()
4  {
5      /* index for our calculations */
6      int   index;
7
8      index = 0;
9
10     /* syntax error on next line */
11     while (index < BIG_NUMBER) {
12         index = index * 8;
13     }
14
15     return (0);
16 }
```

`#define` Statement

- The example cannot be compiled!! Why??

```
1  #define BIG_NUMBER 10 ** 10
2
3  main()
4  {
5      /* index for our calculations */
6      int   index;
7
8      index = 0;
9
10     /* syntax error on next line */
11     while (index < BIG_NUMBER) {
12         index = index * 8;
13     }
14
15     return (0);
16 }
```

`#define` Statement

- The example cannot be compiled!! Why??

```
1  #define BIG_NUMBER 10 ** 10
2
3  main()
4  {
5      /* index for our calculations */
6      int   index;
7
8      index = 0;
9
10     /* syntax error on next line */
11     while (index < BIG_NUMBER) {
12         index = index * 8;
13     }
14
15     return (0);
16 }
```

`#define` Statement

- The example cannot be compiled!! Why??

```
1  #define BIG_NUMBER 10 ** 10
2
3  main()
4  {
5      /* index for our calculations */
6      int   index;
7
8      index = 0;
9
10     /* syntax error on next line */
11     while (index < BIG_NUMBER) {
12         index = index * 8;
13     }
14
15     return (0);
16 }
```

`#define` Statement

- The example cannot be compiled!! Why??

```
while (index < 10 ** 10)
```

```
1  #define BIG_NUMBER 10 ** 10
2
3  main()
4  {
5      /* index for our calculations */
6      int   index;
7
8      index = 0;
9
10     /* syntax error on next line */
11     while (index < BIG_NUMBER) {
12         index = index * 8;
13     }
14
15     return (0);
16 }
```

`#define` Statement

- The example cannot be compiled!! Why??

```
while (index < 10 ** 10)
```

** is an illegal operator, cause syntax error!!

```
1  #define BIG_NUMBER 10 ** 10
2
3  main()
4  {
5      /* index for our calculations */
6      int   index;
7
8      index = 0;
9
10     /* syntax error on next line */
11     while (index < BIG_NUMBER) {
12         index = index * 8;
13     }
14
15     return (0);
16 }
```

`#define` Statement

`#define` Statement

- Example: first.c
  - The example generates the answer 47 instead of the expected answer 144. Why??

`#define` Statement

- Example: first.c
  - The example generates the answer 47 instead of the expected answer 144. Why??

```c
#include <stdio.h>

#define FIRST_PART      7
#define LAST_PART       5
#define ALL_PARTS       FIRST_PART + LAST_PART

int main() {
    printf("The square of all the parts is %d\n",
            ALL_PARTS * ALL_PARTS);
    return (0);
}
```

`#define` Statement

`#define` Statement

- C allows you to run your program through the preprocessor and view the output

`#define` Statement

- C allows you to run your program through the preprocessor and view the output
- command: **gcc -E codes.c**

`#define` Statement

- C allows you to run your program through the preprocessor and view the output
- command: **gcc -E codes.c**
- Previous example

`#define` Statement

- C allows you to run your program through the preprocessor and view the output
- command: **gcc -E codes.c**
- Previous example

```
int main() {
    printf("The square of all the parts is %d\n",
            7 + 5 * 7 + 5);
    return (0);
}
```

`#define` Statement

- C allows you to run your program through the preprocessor and view the output
- command: **gcc -E codes.c**
- Previous example

```
int main() {
    printf("The square of all the parts is %d\n",
            7 + 5 * 7 + 5);
    return (0);
}
```

`#define` Statement

- Example: max.c
  - An expected error!! Why??

```c
#include <stdio.h>

#define MAX =10

int main()
{
    int  counter;

    for (counter = MAX; counter > 0; --counter)
        printf("Hi there\n");

    return (0);
}
```

`#define` Statement

- Example: max.c
  - An expected error!! Why??

```
1  #include <stdio.h>
2
3  #define MAX =10
4
5  int main()
6  {
7      int  counter;
8
9      for (counter = MAX; counter > 0; --counter)
10         printf("Hi there\n");
11
12     return (0);
13 }
```

`#define` Statement

- Example: max.c
  - An expected error!! Why??

```c
#include <stdio.h>

#define MAX =10

int main()
{
    int  counter;

    for (counter = MAX; counter > 0; --counter)
        printf("Hi there\n");

    return (0);
}
```

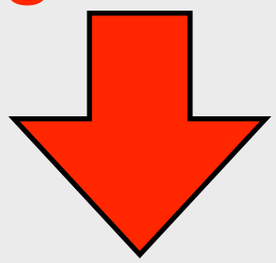
after gcc -E max.c

`#define` Statement

- Example: max.c
  - An expected error!! Why??

```
1  #include <stdio.h>
2
3  #define MAX =10
4
5  int main()
6  {
7      int  counter;
8
9      for (counter = MAX; counter > 0; --counter)
10         printf("Hi there\n");
11
12     return (0);
13 }
```

after gcc -E max.c



`#define` Statement

- Example: max.c
  - An expected error!! Why??

```c
#include <stdio.h>

#define MAX =10

int main()
{
    int  counter;

    for (counter = MAX; counter > 0; --counter)
        printf("Hi there\n");

    return (0);
}
```

after gcc -E max.c

```c
int main()
{
    int counter;

    for (counter = =10; counter > 0; --counter)
        printf("Hi there\n");

    return (0);
}
```

`#define` Statement

- Example: max.c
  - An expected error!! Why??

```c
#include <stdio.h>

#define MAX =10

int main()
{
    int  counter;

    for (counter = MAX; counter > 0; --counter)
        printf("Hi there\n");

    return (0);
}
```

after gcc -E max.c

```c
int main()
{
    int counter;

    for (counter = =10; counter > 0; --counter)
        printf("Hi there\n");

    return (0);
}
```

`#define` Statement

- Example: max.c
  - An expected error!! Why??

```c
#include <stdio.h>

#define MAX =10

int main()
{
    int  counter;

    for (counter = MAX; counter > 0; --counter)
        printf("Hi there\n");

    return (0);
}
```

after gcc -E max.c

```c
int main()
{
    int counter;

    for (counter = =10; counter > 0; --counter)
        printf("Hi there\n");

    return (0);
}
```

Common error!!

`#define` Statement

- Example: size.c
  - Why does 10 - 2 = 10??

```c
#include <stdio.h>

#define SIZE    10;
#define FUDGE   SIZE -2;
int main()
{
    int size;/* size to really use */

    size = FUDGE;
    printf("Size is %d\n", size);
    return (0);
}

```

`#define` Statement

- Example: size.c
  - Why does 10 - 2 = 10??

```c
#include <stdio.h>

#define SIZE    10;
#define FUDGE   SIZE -2;
int main()
{
    int size;/* size to really use */

    size = FUDGE;
    printf("Size is %d\n", size);
    return (0);
}

```

`#define` Statement

- Example: size.c
  - Why does 10 - 2 = 10??

```c
#include <stdio.h>

#define SIZE    10;
#define FUDGE   SIZE -2;
int main()
{
    int size;/* size to really use */

    size = FUDGE;
    printf("Size is %d\n", size);
    return (0);
}

```

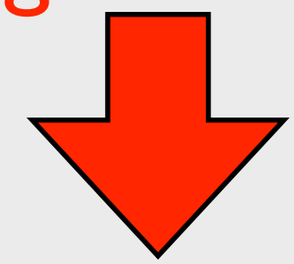
after gcc -E max.c

`#define` Statement

- Example: size.c
  - Why does 10 - 2 = 10??

```c
#include <stdio.h>

#define SIZE    10;
#define FUDGE   SIZE -2;
int main()
{
    int size;/* size to really use */

    size = FUDGE;
    printf("Size is %d\n", size);
    return (0);
}

```

after gcc -E max.c



`#define` Statement

- Example: size.c
  - Why does 10 - 2 = 10??

```c
#include <stdio.h>

#define SIZE    10;
#define FUDGE   SIZE -2;
int main()
{
    int size;/* size to really use */

    size = FUDGE;
    printf("Size is %d\n", size);
    return (0);
}

```

after gcc -E max.c

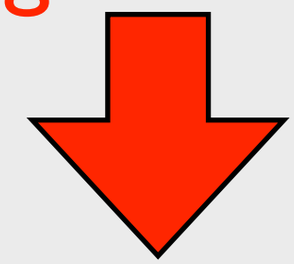
```c
int main()
{
    int size;

    size = 10; -2;;
    printf("Size is %d\n", size);
    return (0);
}
```

`#define` Statement

- Example: size.c
  - Why does 10 - 2 = 10??

```c
#include <stdio.h>

#define SIZE    10;
#define FUDGE   SIZE -2;
int main()
{
    int size;/* size to really use */

    size = FUDGE;
    printf("Size is %d\n", size);
    return (0);
}

```

after gcc -E max.c



```c
int main()
{
    int size;

    size = 10; -2;;
    printf("Size is %d\n", size);
    return (0);
}
```

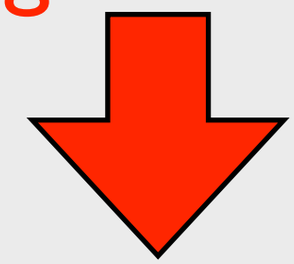
`#define` Statement

- Example: size.c
  - Why does 10 - 2 = 10??

```c
#include <stdio.h>

#define SIZE    10;
#define FUDGE   SIZE -2;
int main()
{
    int size;/* size to really use */

    size = FUDGE;
    printf("Size is %d\n", size);
    return (0);
}

```

Common error!!

after gcc -E max.c



```c
int main()
{
    int size;

    size = 10; -2;;
    printf("Size is %d\n", size);
    return (0);
}
```

`#define` Statement

- Example: die.c
  - Why does it still stop with value = 1?

```
4  #define DIE \
5      fprintf(stderr, "Fatal Error:Abort\n");exit(8);
6
7  int main() {
8      int value;
9
10     value = 1;
11     if (value < 0)
12         DIE;
13
14     printf("We did not die\n");
15     return (0);
16 }
```

`#define` Statement

- Example: die.c
  - Why does it still stop with value = 1?

```
4  #define DIE \
5      fprintf(stderr, "Fatal Error:Abort\n");exit(8);
6
7  int main() {
8      int value;
9
10     value = 1;
11     if (value < 0)
12         DIE;
13
14     printf("We did not die\n");
15     return (0);
16 }
```

`#define` Statement

- Example: die.c
  - Why does it still stop with value = 1?

```
4  #define DIE \
5      fprintf(stderr, "Fatal Error:Abort\n");exit(8);
6
7  int main() {
8      int value;
9
10     value = 1;
11     if (value < 0)
12         DIE;
13
14     printf("We did not die\n");
15     return (0);
16 }
```

after gcc -E max.c

`#define` Statement

- Example: die.c
  - Why does it still stop with value = 1?

```
4  #define DIE \
5      fprintf(stderr, "Fatal Error:Abort\n");exit(8);
6
7  int main() {
8      int value;
9
10     value = 1;
11     if (value < 0)
12         DIE;
13
14     printf("We did not die\n");
15     return (0);
16 }
```

after gcc -E max.c

`#define` Statement

- Example: die.c
  - Why does it still stop with value = 1?

```
4  #define DIE \
5      fprintf(stderr, "Fatal Error:Abort\n");exit(8);
6
7  int main() {
8      int value;
9
10     value = 1;
11     if (value < 0)
12         DIE;
13
14     printf("We did not die\n");
15     return (0);
16 }
```

after gcc -E max.c

```
int main() {
    int value;

    value = 1;
    if (value < 0)
        fprintf(__stderrp, "Fatal Error:Abort\n");exit(8);;

    printf("We did not die\n");
    return (0);
}
```

`#define` Statement

- Example: die.c
  - Why does it still stop with value = 1?

```
4  #define DIE \
5      fprintf(stderr, "Fatal Error:Abort\n");exit(8);
6
7  int main() {
8      int value;
9
10     value = 1;
11     if (value < 0)
12         DIE;
13
14     printf("We did not die\n");
15     return (0);
16 }
```

after gcc -E max.c

```
int main() {
    int value;

    value = 1;
    if (value < 0)
        fprintf(__stderrp, "Fatal Error:Abort\n");exit(8);;

    printf("We did not die\n");
    return (0);
}
```

`#define` Statement

- Example: die.c
  - Why does it still stop with value = 1?

```
4  #define DIE \
5      fprintf(stderr, "Fatal Error:Abort\n");exit(8);
6
7  int main() {
8      int value;
9
10     value = 1;
11     if (value < 0)
12         DIE;
13
14     printf("We did not die\n");
15     return (0);
16 }
```

after gcc -E max.c

```
int main() {
    int value;

    value = 1;
    if (value < 0)
        fprintf(__stderrp, "Fatal Error:Abort\n");exit(8);;

    printf("We did not die\n");
    return (0);
}
```

Common error!!

`#define` vs. `const`

`#define` vs. `const`

- **`const`** is preferred over **`#define`** for following reasons

`#define` vs. `const`

- **`const`** is preferred over **`#define`** for following reasons
  - C checks the syntax of **`const`** statements immediately, whereas **`#define`** is not checked until the macro is used

`#define` vs. `const`

- **`const`** is preferred over **`#define`** for following reasons
  - C checks the syntax of **`const`** statements immediately, whereas **`#define`** is not checked until the macro is used
  - **`const`** uses a C syntax, while the **`#define`** has a syntax all its own

`#define` vs. `const`

- **`const`** is preferred over **`#define`** for following reasons
  - C checks the syntax of **`const`** statements immediately, whereas **`#define`** is not checked until the macro is used
  - **`const`** uses a C syntax, while the **`#define`** has a syntax all its own
  - **`const`** follows normal C scope rules, while constants defined by a **`#define`** continue on forever

`#define` vs. `const`

- Two different usages

`#define` vs. `const`

- Two different usages

```
#define MAX 10 /* Define a value using the preprocessor */
               /* (This definition can easily cause problems) */
```

`#define` vs. `const`

- Two different usages

```
#define MAX 10 /* Define a value using the preprocessor */
               /* (This definition can easily cause problems) */
```

```
const int MAX = 10; /* Define a C constant integer */
                    /* (safer) */
```

`#define` vs. `const`

`#define` vs. `const`

- Complex constants
  - `#define` can only define simple constants, but `const` can define any type of C constant, including things like structured classes

`#define` vs. `const`

- Complex constants
  - **`#define`** can only define simple constants, but **`const`** can define any type of C constant, including things like structured classes

```
struct box {
    int width, height;
};

const struct box pink_box = {1.0, 4.5};
```

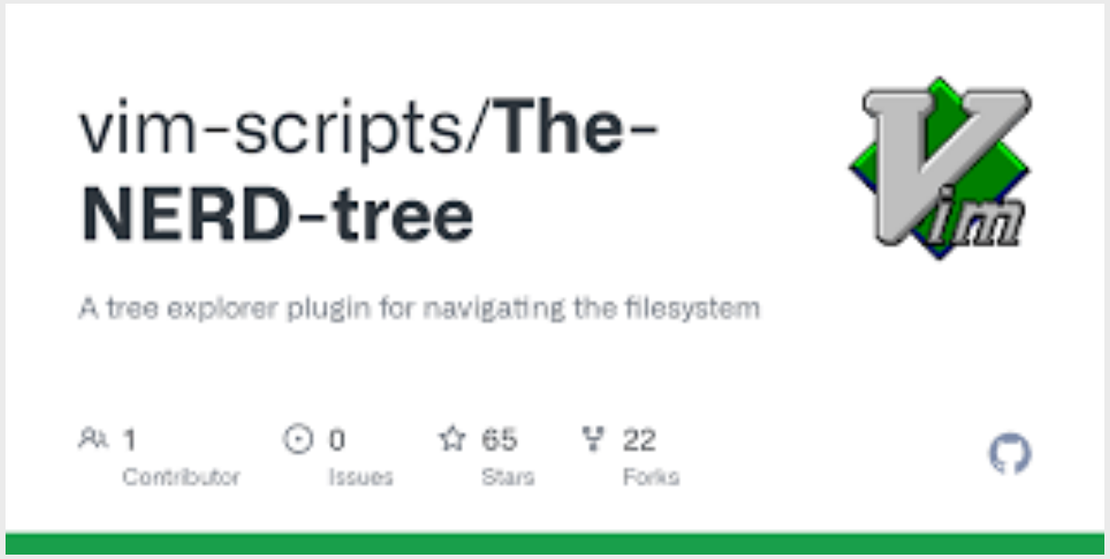
`#define` vs. `const`

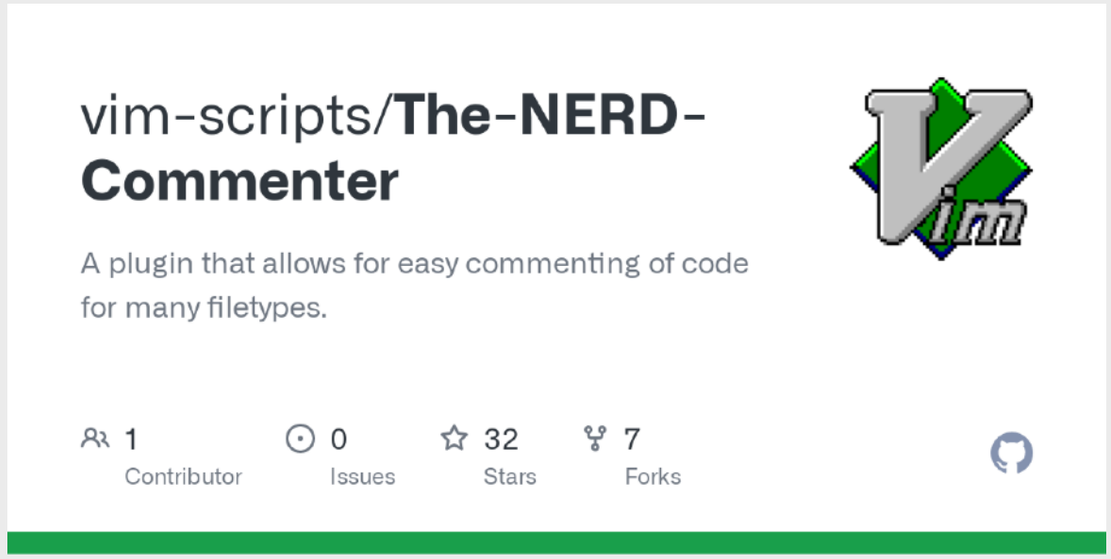
- Complex constants
  - **`#define`** can only define simple constants, but **`const`** can define any type of C constant, including things like structured classes

```
struct box {
    int width, height;
};

const struct box pink_box = {1.0, 4.5};
```

- **`#define`** is essential for things like conditional compilation and other specialized uses

Vim Tips





- Useful Plugins
  - The NERD Tree
  - The NERD Commenter