Statements / Blocks

- An expression becomes a statement when it is followed by a semicolon

```
x = 0;
```

- Braces are used to group declarations and statements into a compound statement

```
{
        x = 0;
        y = 1;
} /* Note: No semicolon after right brace  */
```

if statements

- Shortcut for “equal and not equal to 0” tests

```
if (expression)
```

same as

```
if (expression != 0)
```

else-if

- Consider cascading else-if sequence:

```
if (i == 1)   /* NOTE: Only one will execute */
        statement-1;
else if (i == 2)
        statement-2;
…
else if (i == 49)
        statement-49;
else
        statement-50;    /* Default or "catch-all" */
```

switch

- Also have switch statement **LIKE JAVA**

```
switch (i)  {
  case 1:  statement-1;
      break;
  case 2:  statement-2;
      break;
  .   .   .
  case 49:  statement-49;
      break;
  default:  statement-50;
}
```

Loops –**while** and **for**

- This “for” statement”

```
for ($expr_1$; $expr_2$; $expr_3$)
        statement;
```

- Is equivalent to this “while” statement:

```
$expr_1$;
while ($expr_2$) {
        statement;
        $expr_3$;
}
```

for loop

- Note any part of for loop can be left out.

  for (init; loop-test; increment)
- If init or increment expression is left out, just not evaluated (program must initialize and increment by other means)
- If loop-test is left out, assumes permanently true condition and loops forever. (program must break or goto to end to exit the loop)

do … while

- The do … while tests at the end of the loop

```
do {
        statement(s);
} while (expression);
```

- Executes the statement(s) once even if the “while” loop expression is false upon entry
- Used much less often than “for” and “while”

break and **continue**

- The **break** statement works for:
  - for loop / while loop / do loop and switch.
  - Brings you to end of loop or switch statement ONE LEVEL ONLY.
- The **continue** statement works for:
  - for loop / while loop / do loop, but not switch!
  - It causes next iteration of enclosing loop to begin

Function **Prototype**

- Return type, function name, and ( )

  int foo ( );

  float foo ( );

- Argument List

  List of Types and optionally Variable Names

  int foo (int *, int, float);

  int foo (int array[], int i, float j);

  Output arguments usually listed first

Function Prototype: **Null** Argument

- Must put “void”

```
int foo (void);
```

- Keeps compiler parameter checking enabled
- The following is a **bad** example:

```
int foo ();
x = foo (i, j, k);   /* compiler won’t catch! */
```

Function **Declarations**

- Same as function prototype, except:
  - Must have variable names in argument list
  - Followed by { function statements; } not ;
- Example:

```
int foo (int array[], int i, float j)
{
        function statements;
}
```

C Preprocessor

- Inclusion of other files – usually .h files

  #include “filename.h”
- Simple macro substitution

  #define name substitute text
- Macro substitution with arguments

  #define square(A) ((A)*(A))   enclose in ( )s

  n = square(x) + 1;   →   n = ((x)*(x)) + 1;
- Conditional inclusion

Macros

- Macros do not understand C expressions. They are only doing precise character substitution.
- Macro substitution with arguments – bad example

```
#define square(A)  A*A
```

If you write in program:

```
n = square(p+1);
```

Macro will be replaced by:

```
n = p+1*p+1;
```

Not what you expected

Macros

- Macro must be defined on a single line
- Can continue a long definition to the next line using backslash character (\)

```
#define exchg(t, x, y)  {t  d; d  = x;  x = y;\
y = d;}
```

- The \ simply tells compiler the following line is a continuation of same logical line

Macros

- This macro invocation

```
exchg (char, u, v)
```

will be turned into the following text string (shown one statement per line for clarity)

```
{char d;   /* Note: d is defined locally within block */
d = u;
u = v;
v = d;  }
```

Macros

- Function calls are CALL BY VALUE
- This is NOT true for Macros, because statements within a Macro expansion act like in-line code!
- Frequently used Macro may take more memory than a function, but does not require call/return and stack frames! (Macro will usually execute faster)

Macros

- Substitutions are not done within quotation marks.
- If we want to print out the variable makeup of an expression and then the value (e.g., x/y = 17.2), it doesn't work to do it like this:

```
#define dprint(expr)   printf("expr  = %f\n", expr)
….
dprint(x/y);
```

- We want: “x/y = 17.2” printed but, it expands as:

```
printf ("expr = %f\n", x/y);
```

Macros

- Use a special convention with the # character.

```
#define dprint(expr)   printf(#expr  " = %g\n", expr)
```

- The special form **#expr** means:
  - Do substitution for the macro “expr”
  - Put quotes around result
- Now if we write

```
        dprint(x/y);
```

- Then, this expands as:

```
printf("x/y" " = %g\n", x/y);   /* two strings concatenated */
```

Conditional Inclusion

- Gives control of when precompiler directives such as #define or #include are executed
- It’s done before compilation with conditionals that are meaningful at that time
- Conditionals work for any C statements in their scope, and can be used to drop unneeded code (and save memory space) under some conditions

Conditional Inclusion

#if	(with conditions such as !defined(SYM)
#ifdef SYM	(if SYM is defined)
#ifndef SYM	(if SYM is not defined)
#elif	(else if)
#else	(else)
#endif	(end scope of originating #if)

Conditional Inclusion

- A software release might need different .h header files included for different O/S's
- Before main() we might define:

  ```
  #define SYSV 100
  #define BSD  101
  ```

- For a specific system (say SYSV) we write:

  ```
  #define SYSTEM  SYSV
  ```

Conditional Inclusion

- Define .h file symbolic constants conditionally:

```
#if SYSTEM = = SYSV
#define HDR "sysv.h"
#elif SYSTEM = = BSD
#define HDR "bsd.h"
#else
#define HDR "default.h“
#endif

#include HDR
```

Conditional Inclusion

- We DON’T want to include declarations contained in a abc.h file twice.

```
/* header file: abc.h */
#ifndef XXX_HDR
#define XXX_HDR
…   (contents of abc.h file go here)
#endif   /* XXX_HDR */
```

Recursion

- Any C function may call itself recursively, either directly or after intermediate function calls
- Each time a call is made, a new frame is placed on the stack, containing passed arguments, a position to return to, and automatic variables (if any)

```
int factorial (int n) {
        if (n  > 1) return  n * factorial (n – 1);
        return 1;
}
```

- Programmer must ensure the recursion terminates!

Recursion, Stack Frames

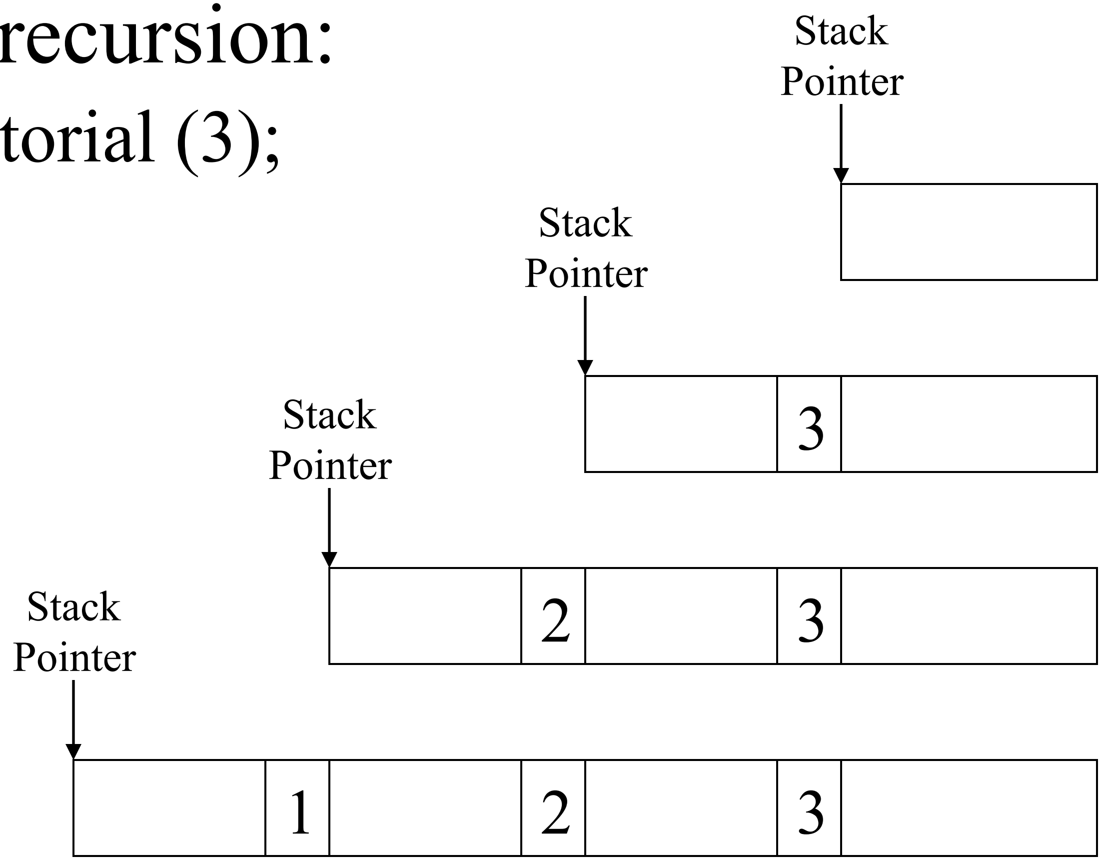
- Stack during recursion:

  int result = factorial (3);

  factorial (3)

  factorial (2)

  factorial (1)



Stack Pointer

Stack Pointer

3

Stack Pointer

2 3

Stack Pointer

1 2 3

Code Example

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

main(){
        int m, n=3;
        m = factorial(n);
 }
int factorial(int k){
        int ll;
        printf("before factorial function: n=%d\n",k);
        ll = (k > 1) ? k*factorial(k-1): 1;
        printf("    after factorial function: n=%d\n", k);
        return ll;
}
```

```
blade64(2)% a.out
before factorial function: n=3
before factorial function: n=2
before factorial function: n=1
    after factorial function: n=1
    after factorial function: n=2
    after factorial function: n=3
```

Recursion, Performance

- Time relative to while/for loops
  - Calling/creating stack frame takes a lot of time
  - Returning/removing stack frame costs time, too
- Memory relative to while/for loops
  - Stack frames eat up memory → need large space!
  - In non-virtual memory system → stack overflow?
- Rarely used in hard real-time and/or embedded systems for these reasons