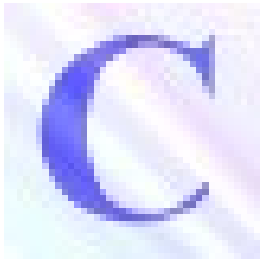
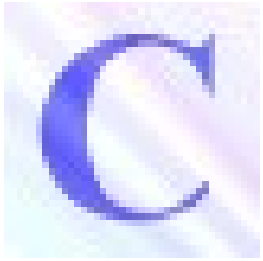


Computer Programming

” What we have to learn to do, we learn by doing.”

Aristotle

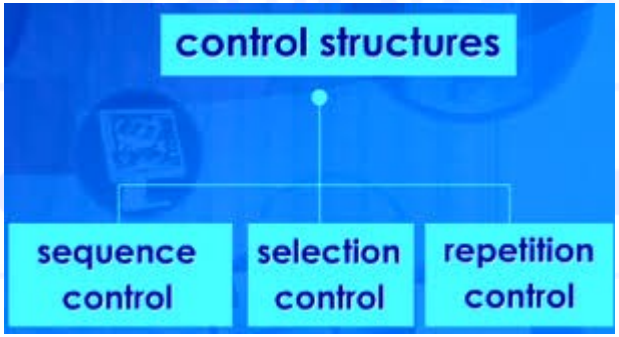


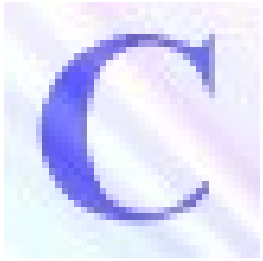
Outline

- Statements
  - Simple statements:
    - *expression*, *empty,* goto, continue, break, return
  - Structured statements
    - if, switch, while, for, do-while
- Preprocessing in C
  - Function vs macro
  - Conditional compilation
  - Constant identifiers

Structured C program

- A structured C program contains only three flow structures:
  - Sequence (composed statements)
  - Alternative (**if** statement) and selection (**switch** statement)
  - Loop structure (statements **while, for, do-while**)





Simple statements. Expression

- Expression format:

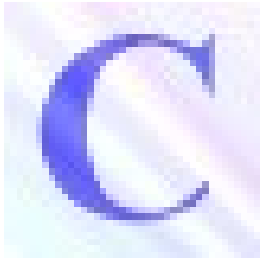
  expression;

  - Semicolon follows *expression*
  - Defined as explained in the previous lecture
  - Used as assignment or function call
- Usage example
  - Find the maximum of two numbers

```
#include <stdio.h>
int main()
{
   int a, b, c;
   printf("\nPlease input two integers, a and b\n");
   scanf("%d %d", &a, &b);
   c=(a > b)? a: b; /* expression statement */
   printf("\nThe maximum of a=%d and b=%d is c=%d\n", a, b, c);
   return 0;
}
```

Simple statements. Expression

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
/* Compute intensity and phase shift in a serial circuit */
int main(int argc, char *argv[])
{
   double V, R, L, f, Z, XL, I, phi;
   printf("\nEffective voltage, V= ");        scanf("%lf", &V);
   printf("\nResistance in Ohms, R= ");       scanf("%lf", &R);
   printf("\nInductance in Henry, L= ");      scanf("%lf", &L);
   printf("\nFrequency in Hertz, f= ");       scanf("%lf", &f);
   XL = 2 * M_PI * f * L;  /* inductance reactance */
   Z = sqrt(R * R + XL *XL); /* impedance */
   I = V / Z; /* intensity in Amperes */
   phi = atan( XL / R ) * 180 / M_PI; /* phase shift in sexagesimal
   degrees */
   printf("\nIntensity I=%6.3f (Amperes)", I);
   printf("\nPhase shift phi=%6.3f (degrees)\n", phi);
   system("PAUSE");
   return 0;
}
```



The empty statement

- An empty statement contains a semicolon only
- Has no effect
- Is used where a statement is needed, but nothing should be executed (e.g. in loops)
- Example

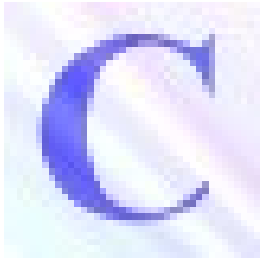
```
for(i = 0, s = 0; i < n; s += a[i], i++);
```

Composite statement

- A *composite statement* is a sequence of statements enclosed between braces, possibly preceded by local declarations
- Used where the computation requires more statements, grouped
- Format

```
{
    declarations;
    statements;
}
```

- Usage examples will be included in the examples which follow



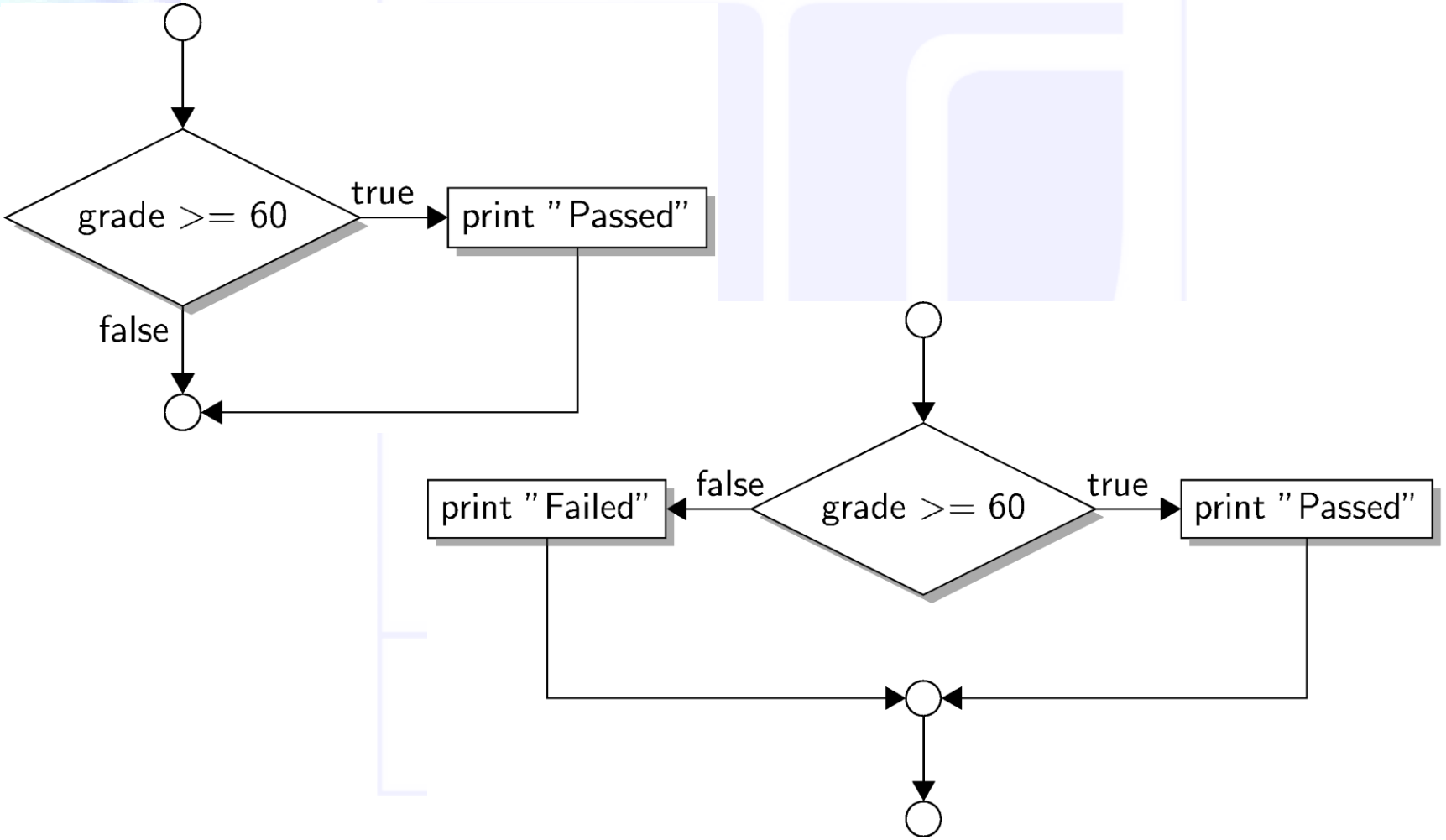
The **if** statement

- Has two formats:
  - **if (expression)**
    statement
  - **if (expression)**
    statement_1
    else
    statement_2
- Effect:
  - **expression** is evaluated
  - if the result is true, **statement** is executed (for the first format) or **statement_1** is executed (for the second format)
  - if the result is false, the statement which immediately follows is executed (first format) or **statement_2** is executed and then the statement which immediately follows (second format)

The **if** statement

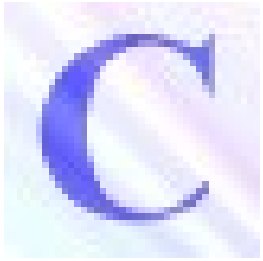
- Notes:
  - The statements (statement, statement_1, statement_2) may contain jump statements which cause control to be passed to other statements than the one following the **if**
  - The **if** may contain other **if** statements. One must be careful with pairing the **else.** If in doubt, use braces

if statement examples



The **if** statement. Usage example: calculate the roots of a second degree equation

```
#include <stdio.h>
#include <math.h>
/* Calculate the roots of the equation a*x^2+b*x+c=0 */
int main(int argc, char *argv[])
{
  float a, b, c, delta, x1, x2;

   printf("\nPlease input coefficients a, b, c\n");
   scanf("%f %f %f", &a, &b, &c);
   if (a !=0 )
   {
      delta = b * b - 4 * a * c;
      if ( delta >= 0 )
      {
         x1 = ( - b - sqrt( delta )) / ( 2 * a );
         x2 = ( - b + sqrt( delta )) / ( 2 * a );
         printf("\nEquation has real roots x1=%g and x2=%g\n", x1, x2);
      }
      else
      {
         x1 = - b / ( 2 * a );
         x2 = sqrt( -delta ) / ( 2 * a );
         printf("\nEquation has complex roots x1=%g-j*%g and x2=%g+j*%g\n",
            x1, x2, x1, x2);
      }
   }
   else printf("\nEquation is not of second degree\n");
  return 0;
}
```

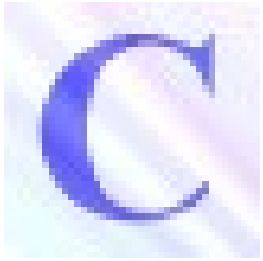


The **switch** statement

- Format:

```
switch ( expression )
{
    case C1:        statements_1;
                    break;
    case C2:        statements_2;
                    break;
    ...
    case Cn:        statements_n;
                    break;
    default:        statements; /* optional */
}
```

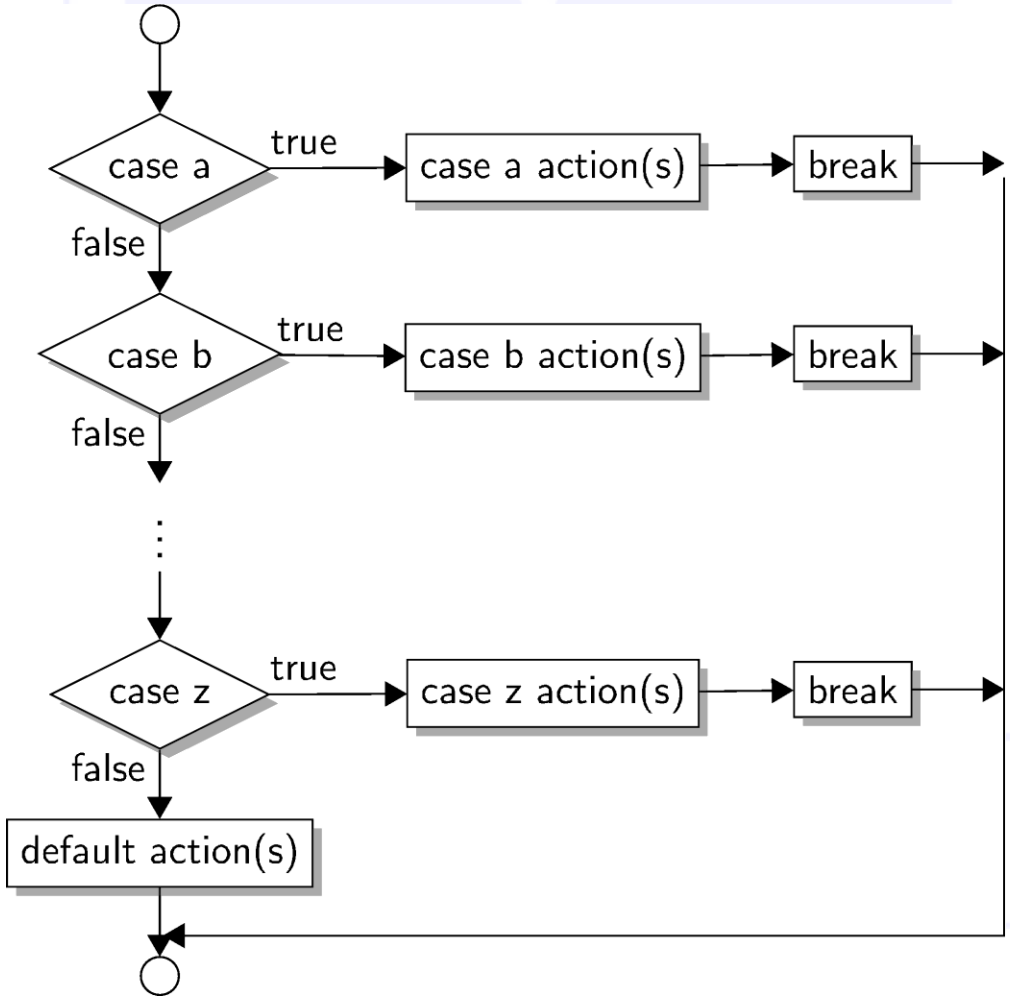
- Effect:
  - **expression** is evaluated
  - if the result matches one of the constants **Ci** then **statements_i** are executed
  - if the result matches no constant, then the statements following the label **default** are executed

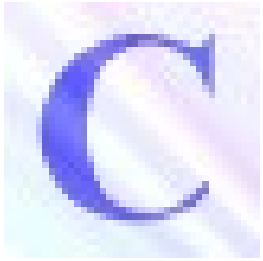


The **switch** statement

- Notes:
  - Label **default** is optional; if the evaluation of expression matches no constant, the **switch** has no effect
  - If no **break** statement is present, then control falls down till a break is met or the **switch** ends
  - A **switch** statement can be replaced by imbricated **if**s

switch Statement





The **switch** Statement. Usage example: evaluate a simple arithmetic expression

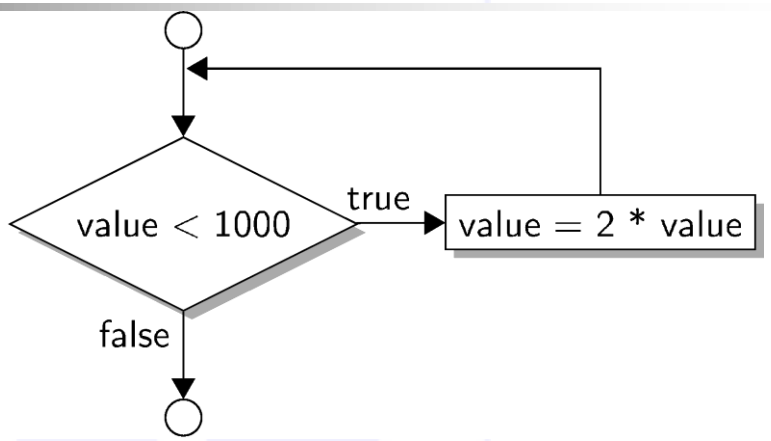
```
#include <stdio.h>
#include <stdlib.h>
/* Evaluate a simple arithmetic expression */
int main()
{
  int operand1, operand2, result;
   char operation;

   printf("Write an arithmetic expression with integers, without spaces\n");
   scanf("%d%c%d", &operand1, &operation, &operand2);
   switch( operation )
   {
        case '+': result = operand1 + operand2;
                                                    break;
        case '-': result = operand1 - operand2;
                                                    break;
        case '*': result = operand1 * operand2;
                                                    break;
        case '/': result = operand1 / operand2;
                                                    break;
        default:  exit(1);
   }
   printf("\n%d %c %d = %d\n", operand1, operation, operand2, result);
  return 0;
}
```

The **while** statement

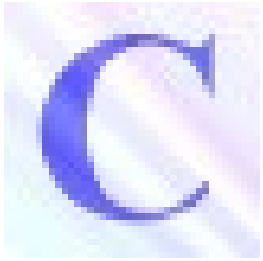
- Format:

  while (expression)

  statement

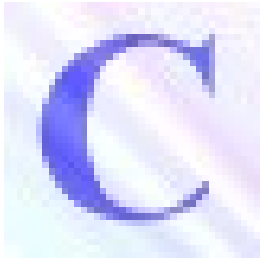


- Effect:
  - Evaluate **expression**
  - If result is true execute **statement;** else execute statement immediately following **while**
- Notes:
  - If result is false upon first evaluation, then **statement** is never executed
  - Within the body of the while statement, statements which change the variables composing expression are necessary



The **while** statement. Usage example: greatest common divisor, smallest common multiple

```
#include <stdio.h>
#include <stdlib.h>
/* Compute the greatest common divisor and the
   smallest common multiple for two numbers */
int main(int argc, char *argv[])
{
   int a, b, a1, b1, gcd, scm, rem;

   printf("Please input number a="); scanf("%d", &a);
   printf("Please input number b="); scanf("%d", &b);
   /* find the gcd */
   a1 = a; b1 = b;
   while ( (rem =a1 % b1) != 0 )
   {
        a1 = b1;
        b1 = rem;
   }
   gcd = b1;
   scm = a * b / gcd;
   printf("a=%d b=%d gcd(a, b)=%d scm(a, b)=%d\n", a, b, gcd, scm);
   return 0;
}
```



The **for** statement

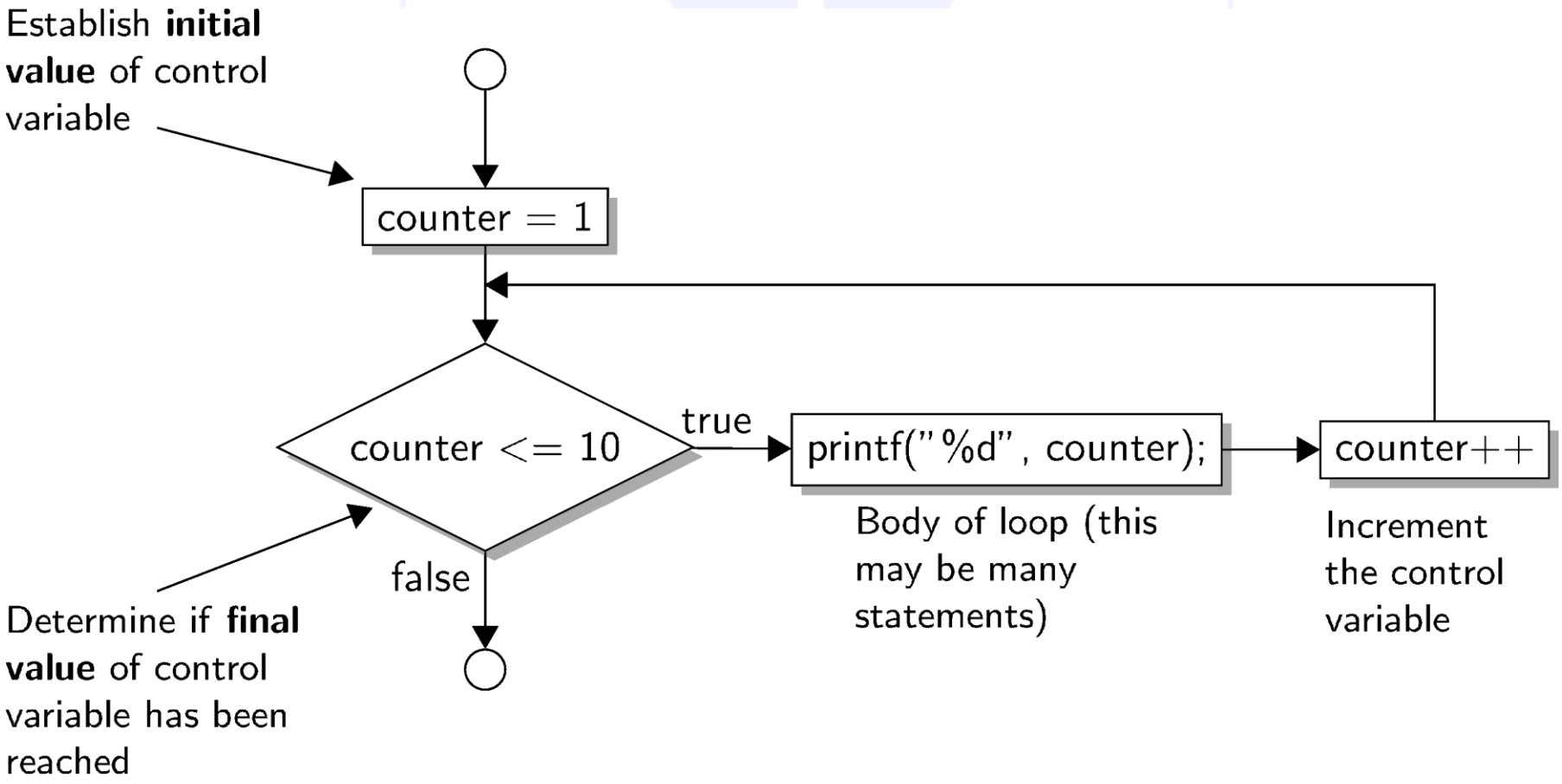
- Format:

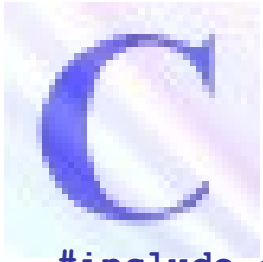
  for (expr1; expr2; expr3)
  statement

  where:
  - **expr1, expr2, expr3** are expressions
  - **statement** is the loop body
- Notes:
  - expr1, expr2, expr3 may be missing but the semicolons must be present
  - An equivalent effect can be obtained using **while:**

```
expr1;
while ( expr2 )
{
   statement;
   expr3;
}
```

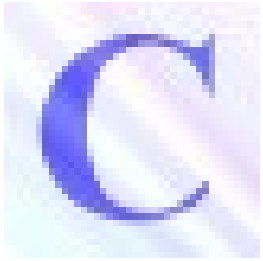
for statement





The **for** statement. Usage example: function approximation using Lagrange's polynomial

```
#include <stdio.h>
#define MAXN 10
/* Function approximation using Lagrange's polynomial */
int main(int argc, char *argv[])
{
  double s, p, x0, x[MAXN], y[MAXN];
   int n, i, j;
   char ch;
   for ( n = 0; n < 1 || n > MAXN; scanf("%d", &n) ) {
        printf("\nPlease input the number of points [<%d], n=", MAXN);
   }
   for ( i = 0; i < n; i++ )      {
        printf("x[%d]=", i+1); scanf("%lf", &x[i]);
        printf("y[%d]=", i+1); scanf("%lf", &y[i]);
   }
   for ( ch = 'Y'; ch == 'Y' || ch == 'y'; ch = getchar() ) {
        printf("Please input a value for x0="); scanf("%lf", &x0);
        for ( s = 0, i = 0; i < n; i++ ) {
                p = 1;
                for ( j = 0; j < n; j++ )
                        if ( i != j ) p = p * ( x0 - x[j]) / ( x[i] - x[j] );
                s += y[i] * p;
        }
        printf("An approximate value for x0=%lf is p0=%lf\n", x0, s);
        printf("Continue [Yes=Y/y No=other character]? ");
   }
  return 0;
}
```



The **for** statement. Usage example: arithmetic average of *n* real numbers

```
#include <stdio.h>
#define NUMELEM 100
int main(int argc, char *argv[])
{
    float a[NUMELEM], average, sum;
    int i, n;

    for ( n = 0; n < 1 || n > NUMELEM; scanf("%d", &n) )
    {
         printf("\nPlease input the number of points [<%d], n=", NUMELEM);
    }
    printf("\nPlease input the elements\n");
    for ( i = 0, sum = 0; i < n; i++ )
    {
         printf("a[%2d]=", i+1); scanf("%f", &a[i]);
         sum += a[i];
    }
    average = sum / n;
    printf("Average=%g\n", average);
  return 0;
}
```

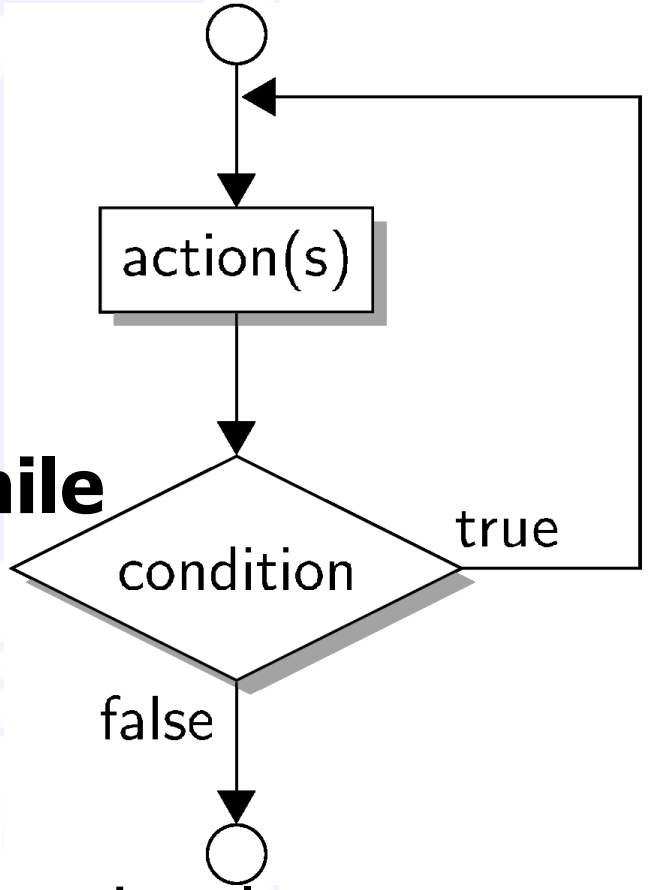
The **do - while** statement

- Format:

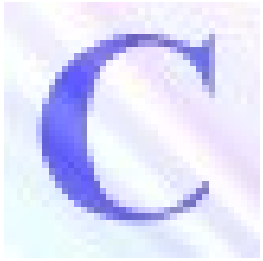
```
do
        statement
while ( expression );
```

- Effect, described in terms of **while**

```
statement;
while ( expression )
        statement;
```

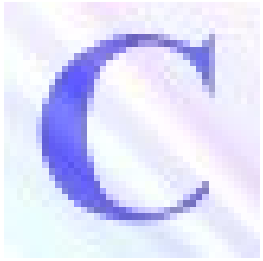


- Note that the loop body is executed at least once



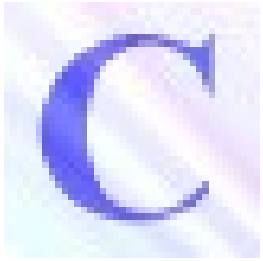
The **do - while** statement. Usage example: arithmetic average of *n* real numbers

```
#include <stdio.h>
#include <stdlib.h>
#define NUMELEM 100
int main(int argc, char *argv[])
{
    float a[NUMELEM], average, sum;
    int i, n;
    n = 0;
    do
    {
      printf("\nPlease input the number of points [<%d], n=", NUMELEM);
      scanf("%d", &n);
    }
    while (n < 1 || n > NUMELEM);
    printf("\nPlease input the elements\n");
    i = 0; sum = 0;
    do
    {
      printf("a[%2d]=", i+1); scanf("%f", &a[i]);
      sum += a[i];
      i++;
    }
    while (i < n);
    average = sum / n;
    printf("Average=%g\n", average);
    return 0;
}
```



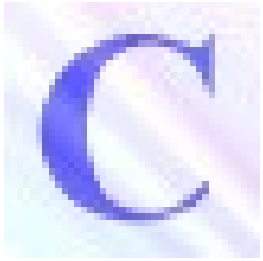
The **do - while** statement. Usage example: greatest common divisor, smallest common multiple

```
#include <stdio.h>
/* Compute the greatest common divisor and the
   smallest common multiple for two numbers */
int main(int argc, char *argv[])
{
   int a, b, a1, b1, gcd, scm, rem;

   printf("Please input number a="); scanf("%d", &a);
   printf("Please input number b="); scanf("%d", &b);
   /* find the gcd */
   a1 = a; b1 = b;
   do
   {
      rem =a1 % b1;
      a1 = b1;
      b1 = rem;
   }
   while ( rem != 0 )
   gcd = b1;
   scm = a * b / gcd;
   printf("a=%d b=%d gcd(a, b)=%d scm(a, b)=%d\n", a, b, gcd, scm);
   return 0;
}
```



The statements **continue** and **break**

- The statements **continue** and **break** can be used only in a loop body (**break** is also used in **switch** to prevent control to flow to the next **case**)
- **continue** causes the current iteration to be abandoned and, for
  - **for** statement, the re-initialization step is executed, and loop expression is reevaluated
  - **while, do-while**: evaluation of expression controlling the loop
- **break** terminates the loop and execution continues with the statement immediately following the loop



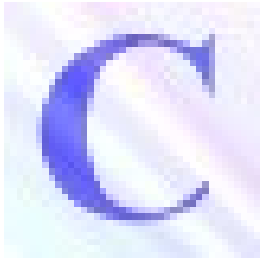
The statements **continue** and **break.** Examples

- A didactic example of replacing the elements of a matrix *not located on the main diagonal*, using **continue**

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
    {
            if ( i==j ) continue;
            a[i][j] *= a[i][j];
    }
```

- A didactic example of replacing the elements of a matrix *situated under the main diagonal*, using **break**

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
    {
            if ( i==j ) break;
            a[i][j] *= a[i][j];
    }
```



The **goto** statement

- **goto** is used for moving the flow control from one point of a function to another *labeled* point of it
- a label is an identifier followed by a colon
- Format for goto:

```
goto label;
```

- Usage example (the sequence which used break):

```
for (i = 0; i < n; i++)
{
        for (j = 0; j < n; j++)
        {
                if ( i==j ) goto quit_inner;
                a[i][j] *= a[i][j];
        }
  quit_inner: ; /* empty statement */
}
```

The **exit** standard function

- The prototype for **`exit`** is found in **`stdlib.h`** and **`process.h`**:

```
void exit( int code);
```

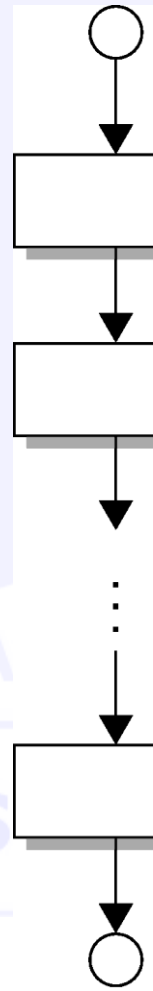
- Is used for forced termination of program execution
- A return code of zero means normal termination. Other values mean errors (code is chosen by programmer)
- Example:

```
n = scanf("%d%d%f%f", &i, &j, &a, &b);
if ( n != 4 ) exit(1); /*forced exit with code 1*/
printf('i=%d j=%d a=%f b=%f\n", j, j, a, b);
```

- **`goto`**, **`continue`**, **`break`** (in loops) and exit cause a program to become unstructured. Thus, their usage is not recommended

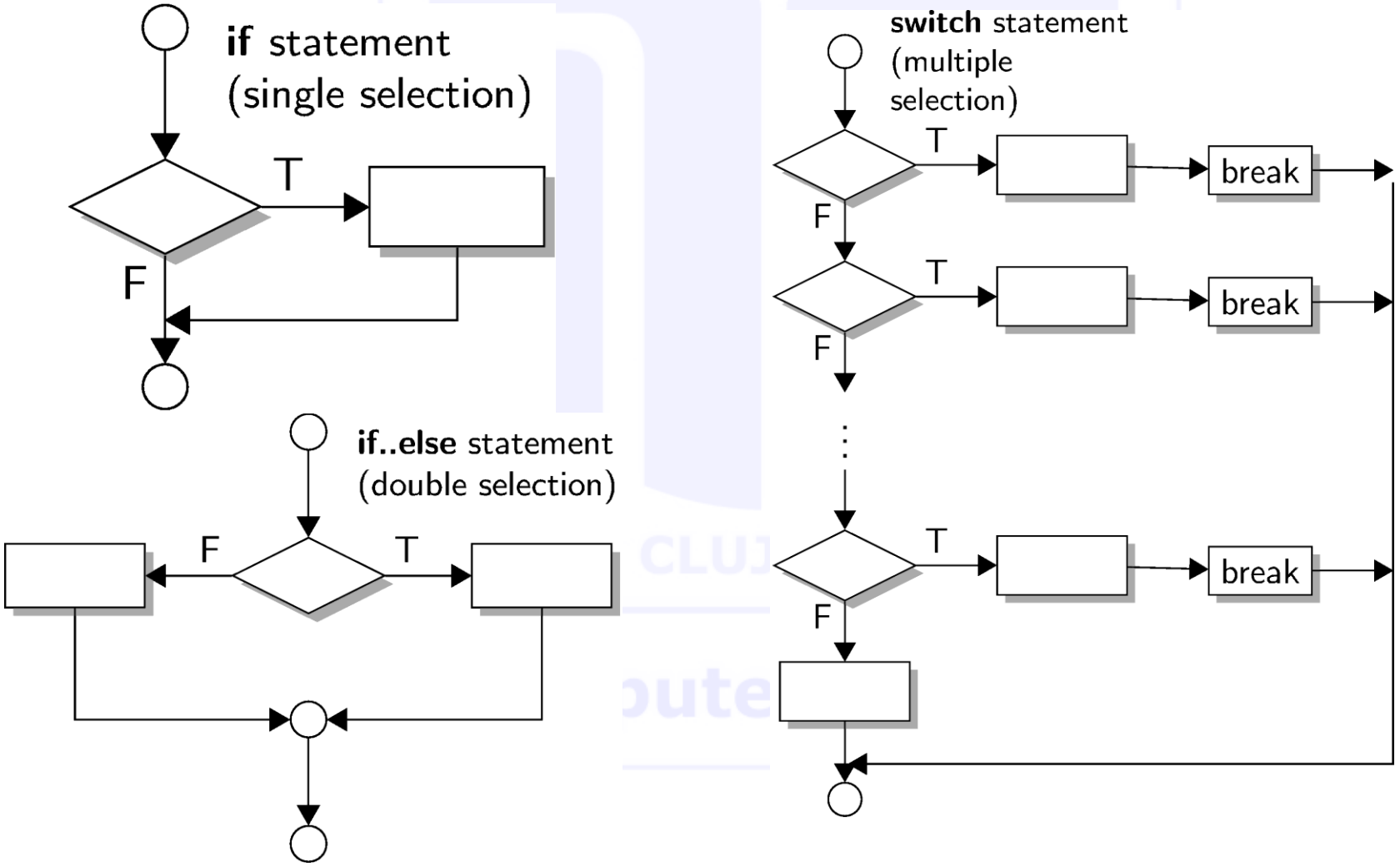
Structured programming

- Allows for only three control structures:
  - Sequence
  - Selection
  - Repetition



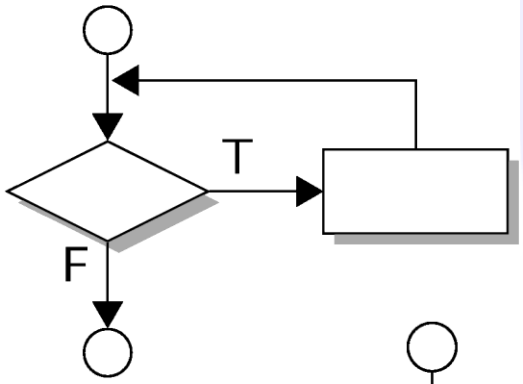
Sequence

Structured programming. Selection

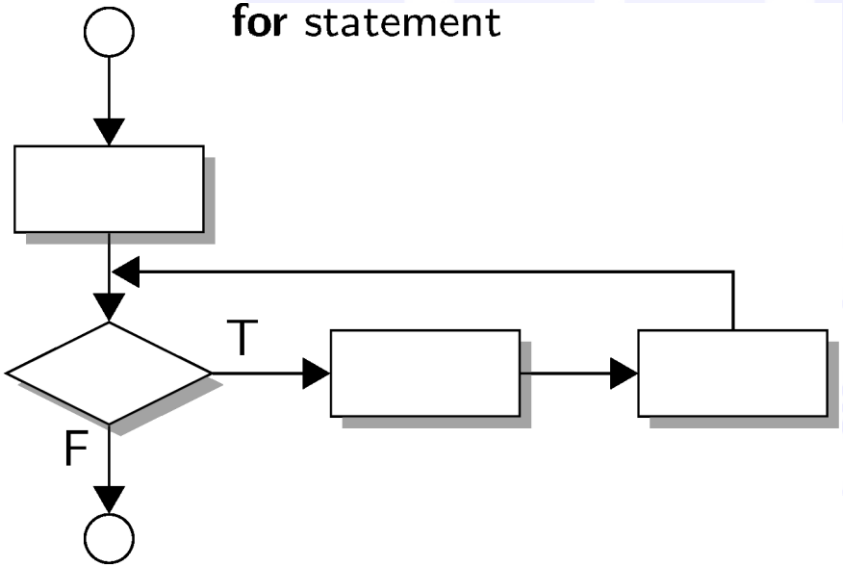


Structured programming. Repetition

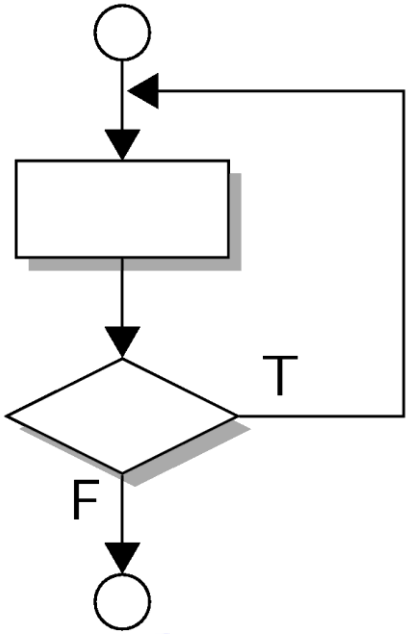
while statement

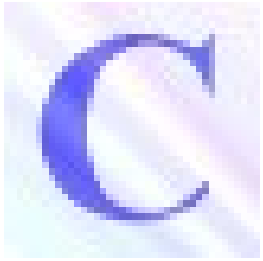


for statement



do..while statement





Preprocessing in C

- Preprocessing occurs before compilation
- Text substitution in source code according to directives
  - Directives are preceded by the sharp character , i.e. **#**
- Preprocessing ensures:
  - Source file inclusion (typically header files)
  - Macro definition and invocation
  - Conditional compilation

Preprocessing in C. File inclusion

- The inclusion directive is specified as:

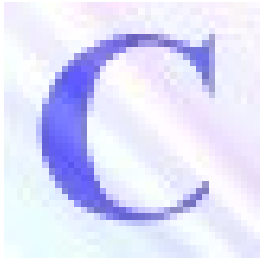
```
#include <file_specifier>
```

or

```
#include "file_specifier"
```

- The directive is replaced by the specified file
- Notes:
  - An included file may contain **include** directives
  - Include directives are placed at the top of a file, for visibility of definitions in the whole file
  - Include directives are widely used for large programs
- Examples

```
#include <stdlib.h>
#include "programmersHeader.h"
```



Preprocessing in C. Symbolic constant and macro definitions

- Directive used is **`#define`**
- Defining a symbolic constant is a special case of macro definition

  `#define name character_sequence`

  - During preprocessing, **name** is replaced by **character_sequence**
  - **character_sequence** may be more than one line long; continuation using `\` as the last character
  - Replacement continues till a directive **#undef name** or the EOF is met

Preprocessing in C. Symbolic constant examples

- Examples:

```
#define ALPHA 20
#define BETA ALPHA+80
```

- Note: with this definitions

```
x=3*BETA
```

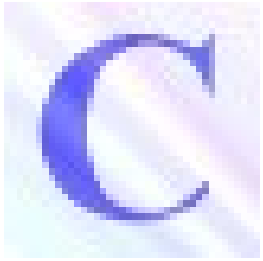
is replaced by

```
x=3*20+80
```

if that is not the intended behavior, **BETA** should be defined as

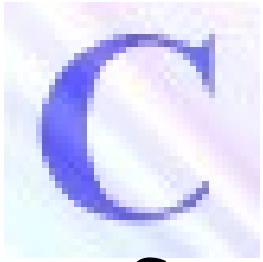
```
#define BETA (ALPHA+80)
```

- Name is replaced by the character sequence which starts with the first non-whitespace



Preprocessing in C. Macro definitions

- A macro definition *resembles* a function definition, i.e.
  `#define name(p1, p2, ..., pn) text`
  where
  - name = macro name
  - `p1, p2, ..., pn` = macro parameters
  - text = substitution text
- Notes
  - Formal parameters are replaced by actual parameters in replacement text
  - **text** may spread over multiple lines. Continuation with `\` as last character
  - Macro substitution is also called **expansion**
  - A symbolic constant is actually a parameter less macro
- Invocation is similar with function invocation, i.e.
  `name(p_actual1, p_actual2, ..., p_actualn)`



Preprocessing in C. Macro definition examples

- Swapping two variables:
  - Definition

```
#define SWAP(vartype, a, b) (vartype t;\
        t=a; a=b; b=t; )
```

  - Invocation

```
SWAP(int, x, y)
SWAP(double, u, v)
```

- Calculate the absolute value
  - Definition

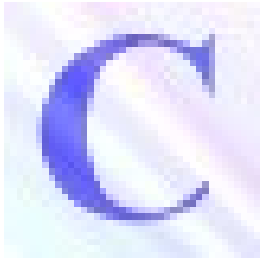
```
#define ABS(x) ( (x) < 0 ? -(x) : (x) )
```

  - Invocation

```
k= ABS(a - b);
```

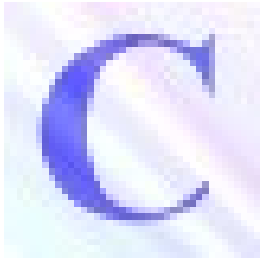
  - Expansion. Note that `x` is *replaced* by `a - b`

```
k= ( (a - b) < 0 ? -(a - b) : (a - b) );
```



Preprocessing in C. Function vs. macro

- Differences between functions and macro definitions
  - **Function invocation** involves a **call** and execution of statements of function body
  - Upon function invocation **parameter type** is also considered
  - **Macro invocation** means **expansion**, i.e. replacement of invocation with macro text. Thus statements in a macro are generated for each invocation and compiled. Hence, macros should be used for small size computations
  - Upon **macro invocation**, a formal parameter is replaced by the character sequence corresponding to the actual parameter. Formal-actual parameter **correspondence is purely positional**
  - Processing time is shorter when using macros (function invocation requires overhead)
- Note. C++ also has **inline** functions, based on a principle similar to macros with the difference that type is also considered



Preprocessing in C. Conditional compilation

- Conditional compilation facilitates development, mainly testing
- Directives for conditional compilation:
  - **#if**:

```
#if expr
      text
#endif
```

```
#if expr
      text1
#else
      text2
#endif
```

where *expr* is a constant expression which can be evaluated by the preprocessor, *text1, text2, text* are portions of source code

Effect: if *expr* is not zero *text* (*text1*) are compiled (not *text2*). Otherwise only *text2* (for second form) and processing continues after **#endif**

Preprocessing in C. Conditional compilation

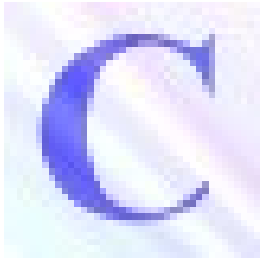
- Directives for conditional compilation:
  - **#ifdef**:

```
#ifdef name
      text
#endif
```

```
#ifdef name
      text1
#else
      text2
#endif
```

where *name* is a constant which is tested if defined; *text1, text2, text* are portions of source code

Effect: if *name* is defined *text* (*text1*) are compiled (not *text2*). Otherwise only *text2* (for second form) and processing continues after **#endif**



Preprocessing in C. Conditional compilation

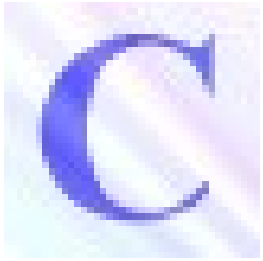
- Directives for conditional compilation:
  - **#ifndef**:

```
#ifndef name
      text
#endif
```

```
#ifndef name
      text1
#else
      text2
#endif
```

where *name* is a constant which is tested if not defined; *text1, text2, text* are portions of source code

Effect: if *name* is undefined *text* (*text1*) are compiled (not *text2*). Otherwise only *text2* (for second form) and processing continues after **#endif**



Preprocessing in C. Conditional compilation

- **#ifdef** and **#ifndef** are used to avoid multiple inclusions. Thus at the beginning of each header file sequences similar to the following can be present (excerpt from **stdio.h**):

```
#ifndef _STDIO_H_
#define   _STDIO_H_
...
#endif /* _STDIO_H_ */
```

- Note. A number of predefined names exist, e.g.:
  - **__DATE__** date of compilation
  - **__CDECL__** function calls follow C conventions
  - **__STDC__** defined if strict C ANSI rules must be followed
  - **__FILE__** full name of currently compiled file
  - **__FUNCTION__** name of current function
  - **__LINE__** number of current line

Preprocessing in C. Declaring constant identifiers

- #error: causes a compilation error with a message given as a parameter to it. E.g.

```
#ifndef __cplusplus
#error "This program should be compiled with C++\
  compilers"
#endif
```

- **Constant declaration:**

```
type const identifier=value;
```

or

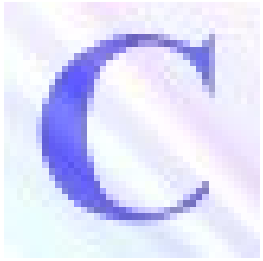
```
const type identifier=value;
```

- **Examples:**

```
int const alpha=10;
double const beta=20.5;
```

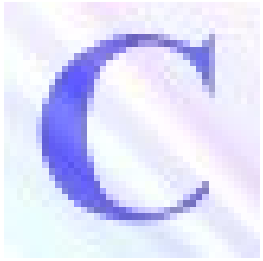
or

```
const int alpha=10;
const double beta=20.5;
```



Reading

- King: chapters 5, 6
- Prata: chapters 5, 6, 7, 16
- Deitel: chapters 3, 4



Summary

- Statements
  - Simple statements:
    - *expression*, *empty,* goto, continue, break, return
  - Structured statements
    - if, switch, while, for, do-while
- Preprocessing in C
  - Function vs macro
  - Conditional compilation
  - Constant identifiers