



# Control Flow (Chapter 3)

CSE 2031  
Fall 2012



# Statements and Blocks (3.1)

- Statement: followed by a semicolon.
- Block
  - enclosed between { and }
  - syntactically equivalent to a single statement
  - no semicolon after the right brace
- Variables can be declared inside *any* block.

# Control Flow Statements

- Similar to Java
- `if - else`
- `else - if`
- `switch`
- `while`
- `for`
- `do - while`
- `break`
- `continue`
- `goto`
- `labels`

if – else

```
if (n > 0)
  if (a > b)
    z = a;
else
  z = b;
```

```
if (n > 0) {
  if (a > b)
    z = a;
}
else
  z = b;
```

if – else – if

```
int binary_search( int x, int v[], int n ) {
    int low, high, mid;
    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low + high)/2;
        if (x < v[mid])
            high = mid + 1;
        else if (x > v[mid])
            low = mid + 1;
        else /* found match */
            return mid;
    }
    return -1; /* no match */
}
```

# switch



```
while ((c = getchar()) != EOF) {  
    switch (c) {  
        case '0': case '1': case '2': case '3': case '4':  
        case '5': case '6': case '7': case '8': case '9':  
            ndigit[c-'0']++;  
            break;  
        case ' ':  
        case '\n':  
        case '\t':  
            nwhite++;  
            break;  
        default:  
            nother++;  
            break;  
    }  
}
```

# Switch

- All cases must be:
  - unique (cannot duplicate cases)
  - constant, e.g. **case 2\*x:** is invalid
- Guidelines
  - avoid deliberate fall-through
  - put a “break” at the end of the switch statement

# while and for Loops

```
while ((c = getchar()) == ' ' || c == '\n'  
      || c == '\t')  
    ; /* skip white space characters */
```

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

```
    ...
```



# do – while



```
do {  
    s[i++] = n % 10 + '0';  
} while ((n /= 10) > 0);
```

Note: the above curly brackets are not necessary. They just make the code more readable.



# continue

Skip negative elements; increment non-negative elements.

```
for (i = 0; i < n; i++) {  
    if (a[i] < 0)        /* skip negative */  
        continue;  
    a[i]++; /* increment non-negative */  
}
```

break



Return the index of the first negative element.

```
...
for (i = 0; i < n; i++)
    if (a[i] < 0) /* 1st negative element */
        break;
if (i < n)
    return i;
...
```

# goto and Labels

Determine whether arrays a and b have an element in common.

```
for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
        if (a[i] == b[j])
            goto found;
/* didn't find any common element */
...
found:
/* got one: a[i] == b[j] */
...
```



# Notes

- Code that relies on `goto` statements is generally harder to understand and to maintain. So `goto` statements should be used rarely, if at all.
- `break` and `continue` should be used only when necessary.

The slide features several decorative light purple circles. One circle is positioned behind the word 'Functions' in the title. Another circle is behind the word 'Program'. A third circle is behind the word 'Structure'. Below the title, there are two more solid light purple circles on the left side, and a fourth circle on the right side that encloses the text 'CSE 2031' and 'Fall 2012'.

# Functions and Program Structure (Chapter 4)

CSE 2031  
Fall 2012



# Program Structure

- C programs are comprised of variables and functions.
- We have discussed variables, expressions and control flow.
- We now want to combine these into a program



# Functions

- A function is a set of statements that may have:
  - a number of arguments, that is values that can be passed to the function
  - a return type that describes the value of this function in an expression



# Defining Functions

- We have seen how to define functions

```
int main() {  
    declarations  
    statements  
}
```

- Defining a function describes its return value, its arguments and provides the code that implements the function

# Returning values

- Two ways to end execution in a function:
  - Let the code fall off the end
  - Use the return keyword
- **return** takes an optional argument - the value to return

```
return 0;
```

or

```
return;
```

# Declaring Functions

- Sometimes we want to use a function without describing how it works
- Declaring a function tells us its return type and arguments but not its code.

```
int putchar(int c);
```

- Like a function definition but with ‘;’ instead of a block

# Declaring Functions

- We can omit argument names

```
int putchar(int) ;
```

- The type of arguments is what matters
- Good practice recommends putting names

void



- “**void**” means “nothing”
- As an argument list: “no arguments”  
`int getchar(void);`
- As a return type: “no return value”  
`void exit(int status);`
- **exit** causes your program to end.

A decorative graphic at the top of the slide consists of a row of five circles. The first circle is solid light purple and overlaps the text 'int main()'. The second circle is hollow with a light purple outline. The third circle is solid light purple. The fourth circle is hollow with a light purple outline. The fifth circle is solid light purple.

int main()?

- Why use: `int main()`  
instead of: `void main()`
- The return value of `main()` is the program's exit status
- In `main()`, `return x;` is the same as `exit(x);`

# Declarations and Return Values

- Declarations (or definitions) are necessary if a function does not return `int`

```
int main() {  
    double atof(char *);  
    printf("%f\n", atof("5.3"));  
}
```

- If we didn't declare `atof()`, `int` would be assumed

Beware!

- Returning a value from a function that should return void is an error
- Returning nothing from a function that should return a value is valid but unpredictable
  - Return value is undefined
- Do neither!



# Scope

- Should be familiar
- Variables only exist within their block:

```
{  
    int x;  
    {  
        int y;  
    }  
    /* y not defined here */  
}
```

# External (or Global) Variables

- What if we want a variable to be available to more than one function?
- Declare it outside of a function:

```
int x;  
  
void add_n_to_x(int n) {  
    x += n;  
}
```

- Visible in all functions

# External Variables

- External variables can be overridden:

```
int x; ← global "x"
```

```
void add_n_to_x(int n) {  
    x += n;  
}
```

```
void set_x_to_m(int m) {  
    int x; ← local "x"  
    x = m;  
}
```

# Multiple Files

- External variables (as well as functions) are visible in other C files

calc.c

```
extern int res;  
void square(int x)  
{  
    res = x*x;  
}
```

main.c

```
int res;  
void square(int);  
  
int main() {  
    square(5);  
    printf("%d\n",  
        res);  
}
```

# How C Programs are Compiled

- C programs go through three stages to be compiled:
  - Preprocessor - handles `#define` and `#include`
  - Compiler - converts C code into binary processor instructions (“object code”)
  - Linker - puts multiple files together and creates an executable program

# How C Programs are Compiled

- When compiling multiple files, all `.c` files are converted to `.o` files
- Then all `.o` files are combined (linked) to make a program.

# How C Programs are Compiled

- You do not have to do this all in one step
- “-c” creates just objects files (“compiles” only)

```
cc -c main.c
```

- Output defaults to “main.o”

```
cc -c calc.c
```

```
cc -o main main.o calc.o
```

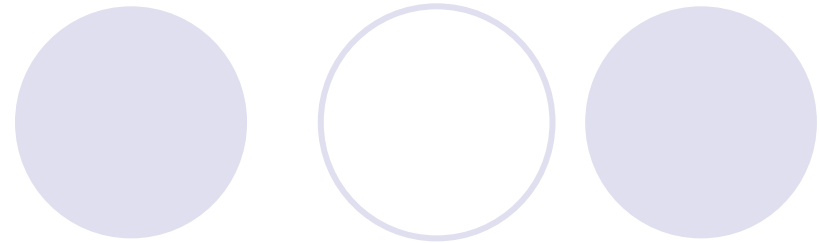
# Hiding Symbols



- By default, all global symbols (functions and global variables) in a source file are visible to the world.
- This is undesirable as it ‘pollutes’ the global namespace and may expose sensitive data.



# Hiding Symbols



- Hide global symbols with **static** keyword  
**static int variable;**
- **static** has a different meaning inside a function
  - makes a variable persistent

# static (Hiding)

`int x;` Visible to other files

`static int y;` Not visible to other files

```
void func1(void) {  
    y++; /* y can still be  
        accessed in this file */  
}
```



# static (Persistent Variables)

- Variables in functions are automatic
  - They are created when the function is called and vanish when the function returns
- External variables are by their nature static.
  - That is they never vanish, value is persistent
- What if we want a variable in a function to be persistent?
  - Declare it **static**



## static (Persistent Variables)

```
int unique_int(void) {  
    static int counter;  
    return counter++;  
}
```

- The value of “counter” is preserved between calls to `unique_int`
- Question: initial value of `counter`?



## static (Persistent Variables)

- Normally variables are not initialized for you (i.e. their values are undefined)
- However, for static variables (and external variables) they are explicitly initialized to zero
- So the first call to `unique_int` returns 0

# The C Preprocessor



- Handles ‘#define’ and ‘#include’
- Removes comments
- Preprocesses C file
  - processes it before compiling it
- Output is C code

#define

- #define defines macros
- Macros substitute one value for another

```
#define IN 1
```

```
state = IN;
```

becomes

```
state = 1;
```

#define

- Macros can also have arguments
- e.g.

```
#define SQUARE(x)  x*x
```

```
y = SQUARE(4) ;
```

becomes

```
y = 4*4 ;
```



#define

- Be careful with arguments

```
SQUARE (5+2)
```

- becomes

```
5+2*5+2      = 17 (!)
```

- Use parentheses defensively, e.g.

```
#define SQUARE (x) ((x) * (x))
```

```
((5+2) * (5+2))      = 49
```

#define

- A macro should only be defined once

```
#define X 5
```

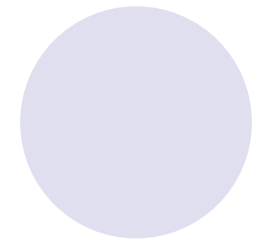
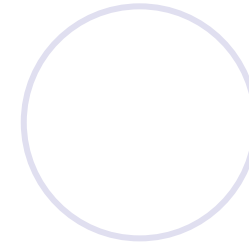
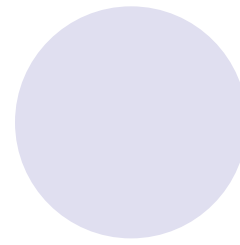
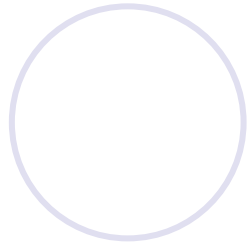
```
#define X 3 -- warning
```

- The name of a macro is important (not its arguments)

```
#define X(x) x
```

```
#define X(x,y) x+y -- warning
```

#define



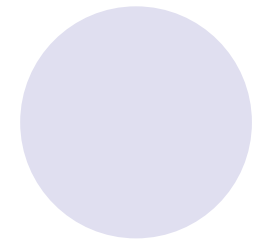
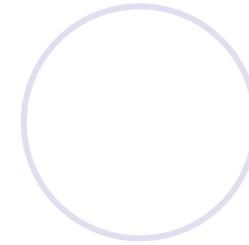
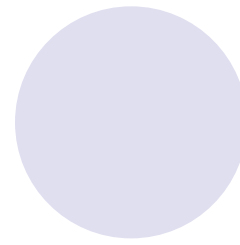
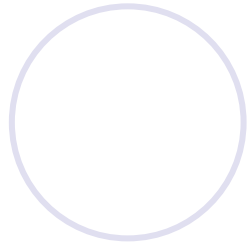
- Macros in substituted values are also evaluated:

```
#define Y Z y
```

```
#define Z z
```

**Y** becomes **z y**

#define



- However - there is no recursion:

```
#define Y Z y
```

```
#define Z Y z
```

Y becomes Y z y

- Any given macro is only substituted once

## ‘#’ operator

- In macros, ‘#’ can be used to make a string

```
#define PRINT(x) printf("%s\n", #x)  
PRINT(hello there);
```

becomes

```
printf("%s\n", "hello there");
```

# ## operator



- ## is the macro concatenation operator
- Puts two names together without space between them

```
#define GLUE(x,y) x##y  
GLUE(foo,bar)
```

becomes

```
foobar
```



**#undef**

- However, what we can define, we can undefine

```
#define X 3
```

- X is replaced with “3”

```
#undef X
```

- X is not replaced

```
#define X 4
```

- X is replaced with “4”

# #if - Conditional Compilation

- We can also use the preprocessor to select what code to compile

```
#if 1
/* This gets compiled */
#else
/* This doesn't */
#endif
```



# #if - Conditional Compilation

- #if takes a constant integer expression and macros can be used

```
#define DEBUG 1
#if DEBUG
printf("debugging message\n");
#endif
```

# #if - Conditional Compilation

- We can also test to see if a macro is defined

```
#if defined(DEBUG)
    printf("debugging\n");
#endif

#if !defined(DEBUG)
    printf("not debugging\n");
#endif
```

# #if - Conditional Compilation

- defined() and !defined() are so common we have constructs for them:

```
#ifdef DEBUG
```

```
    printf("debugging\n");
```

```
#endif
```

```
#ifndef DEBUG
```

```
    printf("not debugging\n");
```

```
#endif
```

# #if - Conditional Compilation

- Often used for platform-specific features

```
#ifdef MACOSX
    /* Mac code */...
#else
    /* Other code */
#endif
```



# #include & Header Files

- `#include` inserts the contents of another file at this point (we talked about this before)
- `#include` is usually used for header files, and header files are really just C code
  - Function declarations
  - Macro definitions
  - External variable declarations
- Do this in one spot so other files can just include the header file

# Multiple Files Revisited

- Introduce “calc.h” as a header file
- Contains declarations for “res” and “square”

calc.h

```
extern int res;  
void square(int x)
```

# Multiple Files Revisited

- Now include this header file in both C files
- Note that we still need to define “res”

```
#include "calc.h"
void square(int x)
{
    res = x*x;
}
```

55

```
#include "calc.h"
int res;    /*!!!*/
int main() {
    square(5);
    printf("%d\n",
           res);
}
```

# Putting It All Together

- A common use of `#ifndef` is to protect header files from being included more than once

calc2.h

```
#ifndef CALC2_H
#define CALC2_H
extern int res;
void square(int x);
#endif
```



# Playing with the C Preprocessor

- Try:

```
cc -E main.c
```

- or with any other C file
- **-E** means “just run the preprocessor”

Next time ...

- Arrays and pointers (chapter 5, C book)