Week 2.2

More C

School of Information Technology and Electrical Engineering
The University of Queensland

Today

More C
- Function pointers
- Casting
- Scope
- Storage classes and qualifiers

Linked lists

Switch statements

Switch = multi-way decision
Example:

```
switch (argc)
{
    case 1:     /* No argument supplied */
        debug=0;
        break;
    case 2:     /* One argument supplied */
        if(strcmp(argv[1], "-debug") == 0)
        {   /* First argument was –debug */
            debug=1;
            break;
        }
        /* else drops through */
    default: /* All other cases */
        printf("Usage: %s [-debug]\n", argv[0]);
        exit(1);
}
```


Switch statements (cont.)

Statement syntax:

```
switch (expression)
{
    case const-expr: statements
    case const-expr: statements
    default:         statements
}
```

`default` is optional
- If omitted and no matching pattern, nothing executed

Execution starts at matching expression
Continues until break statement or end of switch
Usually faster than **`if-else-if-else-if…`**
- May be implemented with table lookup

Expression must be integral type, can't compare strings!

break and continue statements

Can be used inside loops to alter flow of control

break
- terminates execution of innermost **`while`**, **`do`**, **`for`** or **`switch`** statement

continue
- terminates execution of **body** of innermost **`while`**, **`do`** or **`for`** statement and transfers control to end of body
  - i.e. will perform loop again if conditional allows it

Illustration in class

break/continue

Unions

Like a structure (struct), but can only contain *one* of its elements at a time

Example:

```
union U {
    double d;
    char c[2];
    int i;
};
```

Illustration in class

Member access is as for structures

- selection (.)
- indirection (->) for pointers to unions

Programmer has to keep track of which type is stored

union vs struct layout

Function Pointers

Often useful to be able to dynamically choose the function to be called

- e.g. instead of

```
if(i==1) {
    fnOne(…);
} else if (i==2) {
    fnTwo(…);
} else if (i==3) {
    fnThree(…);
} else …
```

- use

```
void (*fnArray[NUM])();
/* Declares fnArray to be
** an array of pointers to
** functions which return
** void. */
fnArray[1] = fnOne;
fnArray[2] = fnTwo;
…
```

- Then call with `fnArray[i](…);`
- or equivalently `(*fnArray[i])(…);`

Note: can combine declaration and initialisation

Some examples

`int (*fp)(int,char*);`

- Declares fp to be a pointer to a function which takes int and char* arguments and returns int

`void (*fp2[10])(double);`

- Declares fp2 to be an array (of size 10) of pointers to functions taking a double parameter and returning nothing

`int (*fp3)();`

- Declares fp3 to be a pointer to a function returning int.
- Argument types unknown and won't be checked by the compiler.
  - Up to programmer to use this correctly.

Exercise (4)

What type is var in each of the following?

```
char *var[];
long var[10];
int **var[10];
void (*var)(int,double);
int *(*var[5])();
void (*var)(int* (*)(int), int);
```

You have 2 minutes

Type casting

Often necessary to convert from one type to another

- Some conversions happen automatically
  - e.g. function arguments, assignment operations, arithmetic expressions
  - Note: doesn't happen for functions like printf which support variable argument types
    - Up to programmer to get it right!
- Other conversions require a **cast**
  - e.g. `dest = (type-name)source;`
- Good to use an explicit cast anyway

Automatic conversions

To	From
Any real type	Any integer type
(void *)	(a) The constant 0 (b) Pointer to object (c) (void*)
Pointer to object	(a) The constant 0 (b) Pointer to compatible object
Pointer to function	(a) The constant 0 (b) Pointer to compatible function

Variable Scope

Scope is the region of a program over which the declaration is visible

Common scopes are

- **file scope**
  - visible from declaration point to end of file
- **function scope**
  - visible from declaration point to end of function
    - includes arguments to function
- **block scope**
  - visible from declaration point to end of block

Variable declarations can be hidden

Example to be given in class

Storage classes

C variable declarations have an *extent* or **storage class**

- **auto**
  - Variable has *local* (automatic) extent, i.e. removed at end of block
  - Permitted within a block only (i.e. not top level)
  - This is the default so rarely seen
- **extern** (for variables or functions)
  - Variable/function is external to all functions, i.e. can be accessed by name by any function
  - Globally accessible – linker must know about the name
  - Must be *defined* once somewhere (can be *declared* anywhere)
- **register**
  - Hint to compiler to put variable in a register, otherwise like auto
- **static** (for variables or functions)
  - Name is only accessible in this file (i.e. not exported to linker)
  - For variables – extent is *static* – variable lasts for life of program

Examples to be presented in class

Type qualifiers

`const`

- Indicates that the value can't change, e.g.
  - `int atoi(const char* str);`
    - characters pointed to by str can't be changed
  - `const int constant_value = 37;`
  - `int * const const_pointer;`
  - `const int * pointer_to_const;`

`volatile`

- Indicates that the value can change in ways not under control of the program
- Often used for interacting with hardware, special memory addresses etc