

Complete (but elementary) C programs

- operators and assignment
- sequence, selection and repetition
- functions
- pointers
- getting input

Operators

- arithmetic:

operator	integer operands	floating point operands
+ addition	$23 + 10 = 33$	$22.5 + 10.3 = 32.8$
- subtraction	$23 - 10 = 13$	$22.5 - 10.3 = 12.2$
* multiplication	$23 * 10 = 230$	$22.5 * 10.3 = 231.75$
/ division	$23 / 10 = 2$	$22.5 / 10.3 = 2.18446(\dots)$
% remainder	$23 \% 10 = 3$	N/A

- classifications:

- unary operator: one operand, e.g. $+(6)$, $-(10.43)$, ~ 112

- binary operator: two operands, e.g. $23 + 10$, $23 \% 10$, $17 \wedge 114$, $0x8 | 0x1$

- ternary operator: can you find one?

- relational operators:

operator	description	example
>	greater than	$x > 100$
>=	greater or equal to	$x >= 20$
<	less than	$x < 100$
<=	less or equal to	$x <= 20$
==	equality	$x == 100$
!=	non-equality	$x != 100$

Assignment Statements

- lvalue = rvalue e.g. `year = 1900 + 99;`
 - value of righthand side is stored in lefthand side
 - rvalue is a variable, constant or expression
 - lvalue is a variable
- an assignment statement has a value (that of the righthand side)

```
year2 = year = 1968;
```

- minimal type checking! e.g. with `int year; float length;`:

```
year = 1968.29;    /* year becomes 1968 */
```

```
length = 2 / 3;    /* length becomes 0.0 */
```

```
length = 2.0 / 3; /* length becomes 0.66... */
```

- `=` is different to the equality operator `==`

```
if (year2 == 1968) ... /* possibly false */
```

```
if (year2 = 2001) ... /* always true */
```

```
if (year2 = 0) ... /* always false */
```

String Assignments and Operations

- string operations require library calls: `month = "Dec" + "ember"` does not work – no in-built string operators!
- we have to use the string library `string.h` (see `man string` on student system)

```
#include <string.h>
```

```
...
```

```
char month[9];
```

```
...
```

```
strcpy(month, "Dec");
```

```
strcat(month, "ember");
```

- in C, strings are implemented as arrays of char
 - individual elements of the string variable `month`:

`month[0], month[1], ..., month[8]`

can be accessed the same way as char variables, e.g. `month[0] = 'd'`;

Precedence, Type Hierarchy and Shorthand Notation

- quiz from lecture C1 !
- logical operators (King table 5.3); bitwise operators (King table 20.1-2)
- arithmetic precedence: King sect 4.1
- general operator precedence: King table 4.2
- data type conversion hierarchy: King sect 7.5
- shorthand operators: e.g. `i += 1;`
- increment and decrement operators: e.g. `i++;`

Sequencing and Selection

- statements are separated by semicolons ";" and are executed in the order they appear in the function (program code)
- a statement may be a (or include) call to a function and the statements of that function are then executed first. e.g.

```
i = i+1;  
z = doit1();  
printf( "%d\n", z );  
...
```

```
int doit() {  
    int z;  
    ...  
    return z;  
}
```

- selection: alternative instruction (sequences) according to some condition(s)

```
if ( i == 1 ) {  
    /* statement sequence 1 */  
} else if ( i > 45 ) {  
    /* statement sequence 2 */  
} else if ( i < -42 ) {  
    /* statement sequence 3 */  
} else {  
    /* statement sequence 4 */  
}
```

Repetition

- three loop constructs in C:
 - while (condition) { ... }
 - for (start; condition; inc/decrement) { ... }
 - do { ... } while (condition);
- use break to exit loops in special cases
- use continue to 'skip' to the bottom of loop
- never use the goto statement!
- e.g. while and for loops

```
int main(void) {
    int i;
    i = 0;
    while (i < 10) {
        printf("%d\n", i);
        i = i+1;
    }
    return 0;
}
```

```
int main(void) {
    int i;
    for (i = 0; i < 10; i = i+1) {
        printf("%d\n", i);
    }
    return 0;
}
```

Functions

- functions must be declared before they are used
 - e.g. `#include <stdio.h>` for `printf ()` definition
 - the definition (i.e. implementation) of the function can come later in the program (or even in a different module)

- a function declaration or 'header' gives its 'type signature'; it consists of:

```
return_type function_name (parameter_list);
```

e.g.:

```
int add(int x, int y);  
int main(void);  
void display(float x, float y);
```

- this gives sufficient information for compiler to tell if function is called properly
 - hence header files are sufficient to use external libraries
- parameters are passed-by-value, so they won't be changed
 - are effectively local variables to the function, except they are initialised when it is called

Example: function.c

```
1:  #include <stdio.h>
2:  int nextYear(int);
3:  int main(void) {
4:      int a, b;
5:
6:      a = 1981;
7:      b = nextYear(a);
8:
9:      if (b == 1982) {
10:         printf("1982\n");
11:     } else {
12:         printf("Help!\n");
13:     }
14:
15:     return 0;
16: }
17:
18: int nextYear(int y) {
19:     int newYear;
20:
21:     y = y+1;
22:     /* changes y, but not a! */
23:
24:     newYear = y;
25:
26:     return newYear;
27: }
```

Pointers

- a pointer is something which may or may not exist – Anonymous
- variables have 5 attributes: name, type, size, value and *address*
 - size is determined by the type
 - given (in bytes) by the `sizeof()` operator; e.g. `sizeof(i)`, `sizeof(int)`
- a pointer is a variable containing a memory address, usually this is the address of another (non-pointer) variable
- we can use the memory address to read or modify the value of the variable the pointer refers to.
 - int * type for pointer to int
 - *x variable pointed to by x
 - &y address of y
- why are these not valid?
 - &100 &'A' &(k+2)

Pointer Example: pointer1.c

```
include <stdio.h>
int main(void) {
    short int *x;    /* x is a pointer to a variable of type int */
    short int a = 2;

    printf("sizeof(a)=%d\n", sizeof(a));
    printf("sizeof(x)=%d\n", sizeof(x));

    x = &a;    /* the address of a is assigned to x */
              /* x points now to where a is in memory */

    *x = *x+1; /* increment the variable that x points */
              /* to, i.e. the variable a */

    printf("a=%d\n", a);
    printf("*x=%d\n", *x);
    printf("x=%p\n", x);    /*note pointer format specification*/
    printf("&a=%p\n", &a);
    printf("&x=%p\n", &x);

    return 0;
}
```

Pointers as Parameters

- passing a pointer as a parameter to a function, allows the function to change the variable the pointer refers to
 - this allows us to work around the restriction that a function cannot modify any variables (other than those local to it)
- we don't change a pointer, only the variable it points to
- remember that all variables are passed-by-value,
 - i.e. the parameter values are copied to local variables when a function is called
 - a memory address is just a value, so a copy of a pointer is just as good as the original pointer itself

Pointer Example: pointer3.c

```
1:  #include <stdio.h>
2:
3:  void nextYear(int *y);
4:
5:  int main(void) {
6:      int thisYear;
7:
8:      thisYear = 1981;
9:      nextYear(&thisYear);
10:
11:     printf("Year: %d\n", thisYear);
12:
13:     return 0;
14: }
```

```
1:  void nextYear(int *y) {
2:
3:     *y = *y + 1 ;
4:
5: }
```

Getting Input – scanf()

- the `stdio` library provides functions to read values from keyboard and other sources (like files)

```
int scanf(const char *format, ...);
```

- the format string contains conversion specifications indicating how many, and what type of values to read
- the subsequent parameters are pointers to the variables in which `scanf()` should store the values e.g. `scanf("%d", &year);`
- `scanf()` uses whitespace characters (return, tab and space) to decide how to divide the input into separate fields
- e.g. `scanf.c`

```
int a, b;  
scanf("%d_%d", &a, &b);  
printf("sum=%d\n", a+b);
```

For Next Lecture: Look at string.c

```
1: #include <stdio.h>
2: int main() {
3:     char name[100];
4:     printf( "Enter your name\n" );
5:     scanf( "%s" , name );
6:     printf( "Your name is %s\n" , name );
7:     return 0;
8: }
```

- why is there no & in this scanf()?
- run the program and input both your first and lastname separated by a space.

What is printed out and why?