C Programming
Memory and Control Flow

Adam Sampson (based on material by Greg Michaelson)
School of Mathematical and Computer Sciences
Heriot-Watt University

Address-of operator: &

- The variable declaration:

  ```
  type name;
  ```

  associates `name` with the address of enough memory to hold the given type

- **`&name`** gives the memory address of the first byte allocated to variable `name` – a **pointer** to `name`
  - On a 32-bit platform, an address needs 4 bytes

scanf

```
scanf("format",addr1...addrN);
```

- Reads keyboard input into memory at specified addresses
- Like `printf`, the values to read are specified by a format string containing format characters
- Typically, `addr`*`i`* is `&name`*`i`* – the address of a variable
- Returns number of items read on (partial) success, or the constant `EOF` on failure

`poly.c`

Evaluating a polynomial: $ax^2 + bx + c$

Evaluating $ax^2 + bx + c$

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int a, b, c, x;
    printf("a: "); scanf("%d", &a);
    printf("b: "); scanf("%d", &b);
    printf("c: "); scanf("%d", &c);
    printf("x: "); scanf("%d", &x);
    printf("%d\n", a*x*x + b*x + c);
    return 0;
}
```

```
$ ./poly
a: 3
b: 8
c: 4
x: 6
160
```

Evaluating $ax^2 + bx + c$

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int a, b, c, x;
    printf("a: "); scanf("%d", &a);
    printf("b: "); scanf("%d", &b);
    printf("c: "); scanf("%d", &c);
    printf("x: "); scanf("%d", &x);
    printf("%d\n", a*x*x + b*x + c);
    return 0;
}
```

Store value read
at address of variable

```
$ ./poly
a: 3
b: 8
c: 4
x: 6
160
```

Indirection operator: *

- **`*expression`** means:
  - Evaluate expression to integer
  - Use integer as address to get value from memory

- Used to access a value indirectly, through a pointer
  - Operand of `*` must be a pointer
  - Result of `*` is the value at the address to which the operand points to

- e.g. in an expression: `*(&name)`
  - Get address associated with `name`
  - Get value from memory at that address

Assignment

```
expression1 = expression2;
```

- Evaluate `expression1` to give an address
  - An **lvalue** – on **l**eft of assignment
- Evaluate `expression2` to give a value
  - An **rvalue** – on **r**ight of assignment
- Put the value in memory at the address

Logic and logical operators

- No boolean type in traditional ANSI C
  - `C99` and newer has `bool`/`true`/`false` in `<stdbool.h>`, but...
- `0` means false; any non-zero value means true
- Unary: `!` not
- Binary:
  - && logical AND
  - `||` logical OR

Comparison operators

- Binary:
  - `==` equality
  - `!=` inequality
  - `<` less than
  - `<=` less than or equal
  - `>` greater than
  - `>=` greater than or equal

Precedence (again)

- `(...)` before
- && before
- `||` before
- `!` before
- comparison before
- arithmetic before
- function call

Blocks

```
{    declarations
     statements
}
```

- Declarations are optional
- Space allocated to declarations
  - on stack or in CPU registers
  - for life of block
- In C99 and later, declarations and statements can be intermixed freely

Iteration: `while`

```
while (expression)
     statement
```

- Evaluate expression
- If non-zero (true) then
    - execute statement (usually a block)
    - repeat from start
- If zero (false) then end iteration
- `break` in statement ends enclosing iteration

`sumav.c`

Sum and average

Sum and average

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int count;
    int sum;
    int n;

    count = 0;
    sum = 0;
```

Sum and average

```
    printf("next> ");
    scanf("%d", &n);
    while (n != 0) {
        count = count + 1;
        sum = sum + n;
        printf("next> ");
        scanf("%d", &n);
    }
    printf("count: %d, sum: %d, average: %d\n",
           count, sum, sum / count);
    return 0;
}
```

Sum and average

```
$ ./sumav
next> 1
next> 2
next> 3
next> 4
next> 5
next> 0
count: 5, sum: 15, average: 3
```

Iteration: `for`

```
for (exp1; exp2; exp3)
     statement;
```

- Execute **`exp1`** once
- Repeatedly:
  - Execute **`exp2`**
  - If true:
    - Execute **`statement`**
    - Execute **`exp3`**
  - If false: exit loop
- All `exp`s and `statement` are optional

Iteration: **for**

```
for (exp1; exp2; exp3)
     statement;
```

… means the same as …

```
exp1;
while (exp2) {
   statement;
   exp3;
}
```

- Execute **exp1** once
- Repeatedly:
  - Execute **exp2**
  - If true:
    - Execute **statement**
    - Execute **exp3**
  - If false: exit loop
- All exps and statement are optional

Using `for`

- `for (exp1; exp2; exp3)` – usually:
  - `exp1` initialises loop control variable
  - `exp2` checks if termination condition is met for variable
  - `exp3` changes control variable

```
int i;
for (i = 0; i < 100; i++) ...
```

(counts 0 to 99)

- In ANSI C, you must declare the variable before `for`; in C99 and later, you can declare it inside exp1:

```
for (int i = 0; i < 100; i++) ...
```

Condition: `if`

```
if (expression)
    statement1
else
    statement2
```

- Evaluate `expression`
- If non-zero (true) then execute `statement1`
- If zero (false) then execute `statement2`
- `else statement2` is optional – empty if omitted

Condition: `switch`

```
switch (expression)
{  case constant1: statements1
   case constant2: statements2
   ...
   default: statementsN
}
```

- Evaluate `expression` to a value
- For first `constant`*i* with same value, execute `statements`*i*
- If no constant matches, evaluate default `statementsN`

Condition: `switch`

```
switch (expression)
{  case constant1: statements1
   case constant2: statements2
   ...
   default: statementsN
}
```

- Only `char` and integer constants are allowed (no strings, etc.)
- `break;` in `statements` jumps to end of `switch`
- If no `break` at end of `statements`*i*,
  will **fall through** to `statements`*i+1*!

Example: guessing game

- Player thinks of a number between 1 and 100
- Computer has to guess number
- Each time, player tells computer if guess is:
  - correct
  - high
  - low
- Computer uses divide and conquer (binary search)
  to halve search space with each guess

Example: guessing game

- Keep track of high and low boundaries
  - Initially high is 100 and low is 1
- Guess number between boundaries
  - If high then set high to guess
  - If low then set low to guess
- At end, output count of guesses

`guess.c`

Guessing game

Guessing game

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int low, high, guess, response, count;

    low = 1;
    high = 100;
    count = 0;
```

Guessing game

```
while (1) {
     guess = (high + low) / 2;
     count = count + 1;
     printf("I guess %d.\n", guess);
     printf("Am I correct (0), high (1)
             or low (2)? ");
     scanf("%d", &response);
     if (response == 0)
         break;
```

`break;` jumps to end of while loop

Guessing game

```
        switch (response) {
        case 1: high = guess; break;
        case 2: low = guess; break;
        default:
            printf("I don't understand %d.\n",
                   response);
            count = count - 1;
        }
    }
    printf("I took %d guesses.\n", count);
    return 0;
}
```

Guessing game

```
$ ./guess
I guess 50.
Am I correct (0), high (1) or low (2)? 1
I guess 25.
Am I correct (0), high (1) or low (2)? 2
I guess 37.
Am I correct (0), high (1) or low (2)? 9
I don't understand 9.
I guess 37.
Am I correct (0), high (1) or low (2)? 1
I guess 31.
Am I correct (0), high (1) or low (2)? 0
I took 4 guesses.
```

Additional reading

- The C Book (this covers ANSI C only, not modern features)
  https://publications.gbdirect.co.uk/c_book
- Chapter 1
  - 1.1 The form of a C program
  - 1.2 Functions
- Chapter 2
  - 2.4 Keywords and identifiers
  - 2.5 Declaration of variables
  - 2.8 Expressions and arithmetic
- Chapter 3
  - 3.2 Control of flow

C lectures

- **Compiling code, program layout, printing/reading data, expressions, arithmetic, memory addresses, control flow, precedence**
- Functions, pointers, file IO, arrays
- Memory allocation, casting, masking, shifting
- Strings, structures, dynamic space allocation, field access
- Recursive structures, 2D arrays, union types