

Introduction to C

School of Information Technology and Electrical Engineering
The University of Queensland

Lecture Outline

- UNIX editors
- Building C programs
- C Programming Language
 - Basic structure of a program
 - Quick overview of some features
 - Arrays
 - Pointers
 - Structures
 - Preprocessor
- Useful UNIX
 - .bashrc, aliases

3

Building C programs

- C program files are typically named `<name>.c`
i.e., lowercase .c extension
- Programs are *compiled* and *linked* to produce an *executable*
- **gcc** command can be used for both compilation and linking
 - **gcc** (used to be GNU C Compiler, now **GNU Compiler Collection**) is a free compiler collection – available for many systems

5

This Week

- Lecture - Tuesday
 - UNIX editors
 - Building C programs on UNIX
 - Introduction to C
 - (May not be new to some of you)
- Lecture - Friday
 - C Preprocessor
 - Makefiles
- Pracs (You will need to spend more than scheduled prac time doing these activities! Use the prac times to seek help from tutors if required.)
 - Learn an editor
 - Learn C - C programming tutorials
 - C programming exercises...

2

UNIX Editors

- It is highly recommended that you learn to use a UNIX text editor
- Two popular editors, suitable for writing programs are
 - **vi** (or vim – “vi improved”)
 - **emacs**
- See pages 57 to 75 of Glass & Ables for a brief introduction to both
- More details, including links to tutorials are on the course website

4

Compilation and Linking

- Explanation in class

6



Use of gcc

- **Compilation** (production of object code)
 - `gcc -c name.c`
 - -c argument means compile but do not link
 - Example above will produce file `name.o`
- **Compilation and Linking in one step**
 - `gcc name.c`
 - Links with standard C library and produces executable named `a.out`
 - `gcc -o executable-name name.c`
 - -o argument specifies the name of the output file

7

Why have separate compilation/linking?

- Large programs are made up of multiple source files
- If change one file, shouldn't have to recompile all the others, just
 - recompile the one that changed
 - link the object files to produced an executable
- Recompiling everything can be a slow process
- The **make** command (and Makefiles) provide an automated mechanism to only recompile files that change
 - More details on Friday

9

C Program – Basic Structure

- Main function name must be **main**
 - This function is executed when program starts
- Blocks of code enclosed by braces { }
- C statements must end with a semicolon ;
- C statements are case sensitive
 - **variable** is not the same as **Variable**
- Comments are within `/* ... */`
 - `//` accepted by newer compilers (C99)
 - Comment is from `//` to end of line
 - We'll use `/* ... */` only

11

Use of gcc (cont.)

- **Linking**
 - `gcc -o executable-name name.o`
- Can give multiple filenames as arguments, e.g.
 - `gcc -o executable-name name1.c name2.c name3.o`
 - Compiles and links as required
- Sometimes need to link with the maths library (-lm) if program uses maths functions
 - `gcc -o executable-name name1.c name2.c ... -lm`

8

C Programming Language

- In this course we expect you to...
 - be able to write C programs from scratch
 - understand the meaning of C programs
 - be able to modify C programs
 - understand how C programs use memory
- Lectures can't teach programming
- You'll need to practice

10

Basic Structure (cont.)

- C program consists of
 - Declarations
 - Function definitions
- Function definitions have
 - Variable declarations
 - Statements

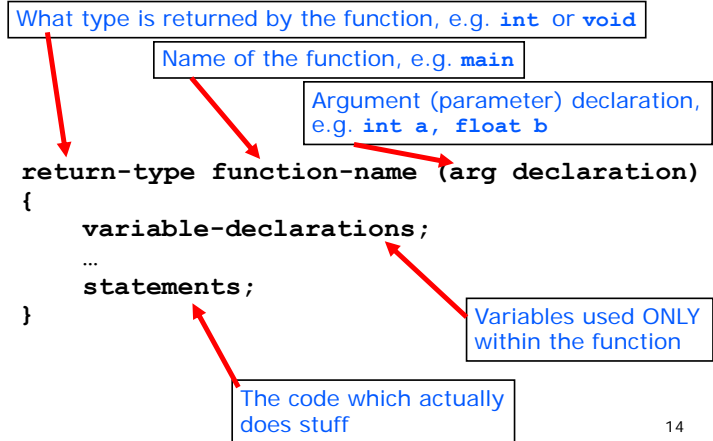
12

Declaring Variables

- Declaration
 - `type-name variable-name, variable-name ...;`
- Examples
 - `char c;` → Single byte
 - `int day;` → Integers (size is machine dependent) Integers can be unsigned or signed (two's complement).
 - `unsigned int count;` → Integers (size is machine dependent) Integers can be unsigned or signed (two's complement).
 - `float expense, income;` → Two single precision floating point numbers
 - `double pi;` → A double precision floating point number
- char, int, float, double are among the data types supported by C
- C does not have a separate `boolean` type (Java does)

13

Function Definitions



14

Function Example

```
/*
   Return the average of two integers
   (result will be rounded towards 0)
*/
int average(int a, int b) {
    int avge;
    avge = (a+b)/2;
    return avge;
}
```

Statements

These are **expressions**.
Outer expression is a **statement**.

The function **returns** a result when finished

15

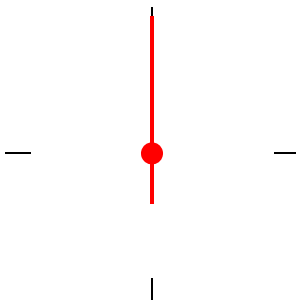
C Constants

- Character constants
 - Use single quotes, e.g. `'a'`, `'b'`, `'1'` etc
 - Some special characters – **backslash escaped**
 - `'\n'` = newline, `'\'` = single quote, `'\t'` = tab, `'\\'` = backslash
- String constants
 - Use double quotes (can include backslash escapes) e.g. `"abc\n\" hello\t"`
- Integer constants
 - Decimal – e.g. `3`, `-27`, `65535`, `+5`
 - Hexadecimal (leading `0x`), e.g. `0x5F`, `0xFFFF`, `0xDEADBEEF`
 - Octal (leading `0`), e.g. `0377` (= 255 decimal)
- Floating point constants
 - Include decimal point (`.`) and/or "e" for exponent
 - Examples: `3.1416`, `-7.`, `6.02e23`, `-5.2e-2`
 - Note `7` is an integer, `7.` is floating point

16

Short Break

- Stand up and stretch



17

Some Operators

- | | |
|--|---|
| ● Binary operators | ● Unary operators |
| <code>+</code> addition | <code>!</code> logical not |
| <code>-</code> subtraction | <code>~</code> one's complement (invert) |
| <code>*</code> multiplication | <code>-</code> two's complement (negate) |
| <code>/</code> division | <code>++</code> increment (prefix or postfix) |
| <code>%</code> remainder (integer) | <code>--</code> decrement (prefix or postfix) |
| <code>></code> greater than | |
| <code>>=</code> greater than or equal | |
| <code>==</code> equal | |
| <code>!=</code> not equal | |
| <code><</code> less than | |
| <code><=</code> less than or equals | |
| <code>&</code> bitwise AND | |
| <code> </code> bitwise OR | |
| <code>^</code> bitwise XOR | |
| <code>&&</code> logical AND | |
| <code> </code> logical OR | |

18

More Operators: Bit-shifting and assignment

- $a \ll b$ means a shifted left by b bits
- $a \gg b$ means a shifted right by b bits
 - What bits are shifted in from the left depends on whether a is signed or not. Do not rely on this.
- $a = b$ means a is assigned the value of b
- $a += b$ is shorthand for $a = a + b$
- Similarly $-=$, $*=$, $/=$, $\%=$, $\&=$, $|=$, $\^=$, $\ll=$, $\gg=$
- Examples
 - $1 \ll 5$ is $1 * 2^5 = 32$
 - $3 \ll 4$ is $3 * 2^4 = 48$
 - $a += 1$ same as $a++$

19

Operator Precedence

- Consider $a + b * c$
- C has strict operator precedence to disambiguate expressions like the above
- Above expression means $a + (b * c)$
- Some operators associate right to left, e.g.
 - $\sim ++ a$ means $\sim (++ a)$
- Most associate left to right:
 - $a - b - c$ means $(a - b) - c$ not $a - (b - c)$

21

Exercise (1)

- What's the result of this code?


```
int a=6;
int b=4;
int c=2;
int d=6;
a/=b++/2+c;
c=c+++c;
d=d+d++;
printf("%d %d %d %d\n", a, b, c, d);
```

(No need to copy this code down – all exercises will be posted to course website)

You have 1½ minutes

24

Postfix/Prefix Increment and Decrement

- Example:

```
int a,b,c,d,e;
a = 4;
b = a++; /* b=a; a=a+1; */
c = --a; /* a=a-1; c=a; */
d = ++a; /* a=a+1; d=a; */
e = a--; /* e=a; a=a-1; */
```

- After these statements, values are $a=4, b=4, c=4, d=5, e=5$

Postfix – change happens after the value used

Prefix – change happens before the value used

20

Operator Precedence and Associativity

Operators	Associativity
<code>() [] -> .</code>	Left to right
<code>! ~ + - ++ -- & * (unary versions)</code>	Right to left
<code>* / %</code>	Left to right
<code>+ -</code>	Left to right
<code><< >></code>	Left to right
<code>< <= > >=</code>	Left to right
<code>== !=</code>	Left to right
<code>& (bitwise and)</code>	Left to right
<code>^ (bitwise xor)</code>	Left to right
<code> (bitwise or)</code>	Left to right
<code>&& (logical and)</code>	Left to right
<code> (logical or)</code>	Left to right
<code>?:</code>	Right to left
<code>= *= /= %= += -= &= ^= = <<= >>= (assignment)</code>	Right to left
<code>,</code>	Left to right

Increasing Precedence ↑

Control Statements

- `if (expression) stmt else stmt`
 - `else` clause is optional
- `while (expression) stmt`
- `do stmt while (expression)`
- `for (init_expr; test_expr; end_expr) stmt`
- Note on expressions:
 - C interprets any 0 value as false, anything else as true
 - (Java has a specific boolean type)
- `stmt` can be replaced by multiple statements enclosed in braces `{ }`

25

Exercise (2)

- What's the result of this code?

```
int a=6;
int b=4;
if(a-b)
    printf("one\n");
else
    printf("two\n");
if(a=b)
    printf("three\n");
else
    printf("four\n");
```

You have 1 minute

Exercise (3)

- What's the result of this code?

```
int a,i;
a=0;
for(i=1;i<6;i++)
    ++a;
printf("%d %d\n", a, i);
a=0;
for(i=0;i<=6;i+=2);
{
    a++;
}
printf("%d %d\n", a, i);
```

You have 2 minutes

Useful UNIX

- Shell settings
- [To be presented in class]

32

For Loop Equivalent Code

```
for (init_expr; test_expr; end_expr) statement1;
```

is equivalent to:

```
init_expr;
while (test_expr) {
    statement1;
    end_expr;
}
```

Statement can be replaced by multiple statements enclosed in braces

- Any or all of the expressions can be empty
- Can use comma to separate multiple expressions:
`for (i=0, j=0; i<10; i++, j+= 4)`

28

Break

31

Useful UNIX

33

Function Return Values

- If no return type is given, C assumes **int**
- Where no return value is desired, the keyword **void** can and should be used
- It is an error to return the wrong type
- Good idea to **prototype** a function before it is used
 - Especially if used before being defined, or defined in another file
 - Header files (.h files) contain prototypes for library functions
- A prototype is like a call to a procedure, but appears outside any procedure
- Has no procedure body

34

Function Prototypes

```

int get_voltage(void);           /* Prototype for 1st proc */
void disp_voltage(int voltage); /* Another prototype */

main() {                         /* main does not need a prototype */
    ...
    v = get_voltage();
    ...
    disp_voltage(v);
}

int get_voltage(void) {          /* Actual body for 1st */
    return inp(...);           /* procedure */
}

void disp_voltage(int voltage) {
    printf(...);
}

```

35

Where do variables live?

```

int a;
float b;

unsigned int max(unsigned int n1,
                 unsigned int n2,
                 unsigned int n3)
{
    int max;
    max=n1;
    if (n2 > max) max=n2;
    if (n3 > max) {
        max=n3;
    }
    return max;
}

```

“global” variables are allocated fixed addresses in memory

function variables are allocated memory every time the function is called. Memory is reclaimed at end of function.

36

Arrays

- Declaring an array


```
type variable-name[size];
```

 - Examples:


```
char message[16];
int values[10];
```
- Accessing elements within an array


```
variable-name[index]
```

 - index = 0 ... size-1
 - This is called zero-based indexing
 - Examples:


```
message[0] = 'c';
values[9] = values[8]++;
```

37

Strings

- A string in C is an array of characters
 - End of string indicated by null character

Arrays in Memory

- [To be presented in class]

Array Initialisation

- Arrays can be initialised at declaration, e.g.
 - `int values[9] = {3, 1, 4, 1, 5};`
 - if variable is global (static) – remaining elements initialised to 0
 - if variable is local (automatic) – remaining elements are uninitialised
- Size can be omitted if array is initialised, e.g.
 - `int a[] = {2,3,5,7};`
 - length is 4 in this case

40

Pointers

- C has concept of pointers
- Pointer declaration
 - `type * variable-name;`
 - `variable-name` is a pointer to something of given `type`
 - How? – pointer variables store memory addresses
- Example:

```
char a, b;
char *ptr;

ptr = &a;
b = *ptr;
```

Can write these on one line:
`char a,b,*ptr;`

& is address-of operator –
creates a pointer

* is indirection operator –
returns value pointed to

- Figures to be drawn in class

42

Pointers and Arrays

- Array name can be treated as a pointer to the first element
 - i.e. address of first element
- Example:


```
int a[10];
int *ptr;

/* following statements are same */
ptr = a;
ptr = &a[0];
```

44

Initialising String Arrays

- [To be presented in class]

41

Operations on Pointers

- Addition/subtraction operations on pointers work in multiples of the size of the object being pointed to
- Example


```
int a[10];
int *ptr;

/* following statements are same */
ptr = a+5;
ptr = &a[5];
```

45

Traversing an Array

- Two examples of clearing an array

- Using **index**:

```
float a[10];
int index;
for(index=0; index<10; index++) {
    a[index] = 0.0;
}
```

- Using **pointer**:

```
float a[10], *ptr;
for(ptr=a; ptr < a+10; ptr++) {
    *ptr = 0.0;
}
```

Adding one to an array pointer makes it point to next element in array

46

Example Function

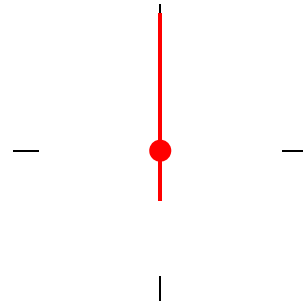
Example Function

- Copying a string – can use index or pointer
- [One version to be presented in class, try writing the other yourself]

47

Short Break

- Stand up and stretch



48

Function Arguments

- Arguments passed to functions are copied (**passed by value**)
- Changes made within function don't affect original arguments
- Example:

```
void swap(int n1, int n2) {
    int tmp;
    tmp = n1;
    n1 = n2;
    n2 = tmp;
}
void main() {
    int a,b;
    a = 2;
    b = 3;
    swap(a,b);
    ... /* nothing has happened */
}
```

This doesn't apply when arrays are passed to functions – since only a pointer to the array is passed.

50

Pointers and Functions

- If pass pointers as an argument to function, CAN change value that is pointed to (called **passing by reference**)
 - (The pointer is copied - not the value pointed to)
- Example:

```
void swap(int *n1, int *n2) {
    int tmp;
    tmp = *n1;
    *n1 = *n2;
    *n2 = tmp;
}
void main() {
    int a,b;
    a = 2;
    b = 3;
    swap(&a, &b);
    ... /* a and b will be swapped */
}
```

51

Structures

- Like a class or record – groups several elements (called members or components) together:

```
/* Structure definition */
struct Time {
    int hour;    /* 0 - 23 */
    int minute; /* 0 - 59 */
    int second; /* 0 - 59 */
};
struct Time time; /* variable decl. */
```

This is the type  This is the variable name

- Members can be accessed using . (selection) operator


```
time.hour = 11;
minutes = time.hour*60 + time.minute;
```

52

Structures and Pointers

- Pointers can point to structures
- Indirection operator ->
- [Code examples to be given in class]

54

Things To Do This Week

- Learn a UNIX text editor
 - vi
 - emacs
- Learn C
 - Do C programming tutorials
 - Work on C programming exercises
 - Start on assignment one (later in the week)
- Friday lecture
 - Let me know if there are questions you want answered
 - C preprocessor
 - Makefiles

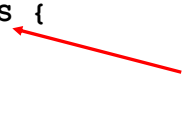
56

typedef

- Can define new names for types
 - Often used with structures, but can be used for any type

```
typedef struct S {
    int a;
    int b;
} s1type;
```

S can be omitted from definition (have to use s1type name)



- s1type is exactly the same as struct S

```
typedef int boolean;
```

- Defines "boolean" to be a synonym for int

53

Structures and Pointers

55

57