

Whiteboard

目录

1. Revision
 1. Style Guide
 1. Do not use 'magic'
 2. Deviation from the style guide: the switch statement
 2. Obfuscation
 1. IOCCC
 3. Assignments and conditions are expressions
 4. Types
 1. Arrays
 1. Array initialisation
 2. Array passing to functions
 3. Arrays of multi-dimensional
 4. Multi-dimension array passing to functions
 2. Structs
 1. Typedefs
 2. Accessing data in structs
 5. Command-Line Arguments
 1. Command-line argument processing example 1
 2. Command-line argument processing example 2
 6. Makefile
 1. 1. special macros
 2. touch the timestamp
 7. Asserts
 1. Does assert help handle errors?
 2. NDEBUG and assert

Revision

Style Guide

Follow the  First-Year C Programming Style Guide

- *How strictly should you follow the style guide?*
 - strictly, except:
 - *switch* statements, which use *break* statements
 - layer of common sense
 - multiple returns (sometimes necessary but don't abuse)
 - variable declarations (generally at start, except local variables, loop variables ...)
 - 'standard' makes marking easy, 'non-standard' may annoy the marker
 - definitely forbidden
 - global and static variables
 - magic numbers
 - programs in lectures sometimes deviate from the guide
 - no description
 - no prototype
 - compacted to fit on screen
 - sometimes 'magic numbers' appear (to keep code to a minimum)

C has a reputation for unreadable/unmaintainable (unmarketable!) code

- avoid *obfuscating* your program

Do not use 'magic'

Magic numbers are symbolic constants that appear 'magically' in the code

- avoid by using a *#define*
- convention is to use capital letters and underscores only

切换行号显示

```

1 #define SPEED_OF_LIGHT 299792458.0    // metres/sec
2 #define SPEED_OF_SOUND 343.0          // metres/sec in dry air, 20
degrees centigrade
3
4 #include <stdio.h>
5 int main(void)
6 {
7     float x = SPEED_OF_LIGHT / SPEED_OF_SOUND;
8     printf ("Light is %f times faster than sound\n", x);
9
10    int y = SPEED_OF_LIGHT / SPEED_OF_SOUND;
11    printf ("Light is %d times faster than sound\n", y);
12 }
```

Output:

```

Light is 874030.500000 times faster than sound
Light is 874030 times faster than sound
```

Deviation from the style guide: the switch statement

A 'multiway' *if-statement* such as:

切换行号显示

```

1 if (var == Constant1) {
2     some statements1;
3 }
4 else if (var == Constant2) {
5     some statements2;
6 }
7 ...
8 else if (var == Constantn) {
9     some statementsn;
10 }
11 else {
12     some statementsn+1;
13 }
```

can be more concisely and readably written as a *switch* statement:

切换行号显示

```

1 switch (var) {
2 case Constant1:
3     some statements1;
4     break;
5 case Constant2:
```

```

6     some statements2;
7     break;
8 ...
9 case Constantn:
10    some statementsn;
11    break;
12 default:
13    some statementsn+1;
14 }

```

Note:

- the *break* is critical; if not present, the execution continues on to the next case.
- if more than one case does the same thing, they can be placed together:
 - e.g.

切换行号显示

```

1  switch(n) {
2  case 1:
3  case 2:
4  case 3: printf("n = 1, 2 or 3\n");
5          break;
6  case 4: printf("n = 4\n");
7  default: printf("no case found\n");
8  }

```

- *but the program above contains a bug: what is it?*

Obfuscation

What does the following program do?

切换行号显示


```

1 // obscure1.c
2 #include <stdio.h>
3 #define __ 5 // do not do this
4
5 int main(void) {
6     int _;
7     for (_=0; _<__; _++) {
8         printf("%d\n", _);
9     }
10    return 0;
11 }

```

- *this is hard to read!*

IOCCC

There is an  International Obfuscated C Code Contest (IOCCC).

- run each year since 1984
- goal is to produce a working C program whose
 - appearance is obscure
 - functionality is impossible to understand
- many categories:
 - *Most in need of wide space*
 - *Most in need of whitespace*

- *Most shifty*
- *Most unstable*
- *Most in need of debugging*
- *Best one-liner*
- ...

2018: Michael Burton (USA), best one liner

Wrote a program **prog.c**:

切换行号显示

```
1 char O,o[];main(1){for(;~1;O||puts(o))O=(O[o]=~(l=getchar()))?
4<(4^1>>5)?l:46:0)?~O&printf("%02x ",l)*5:!O;}
```

Compiling and executing the program:

```
prompt$ gcc prog.c
prompt$ ./a.out < prog.c
63 68 61 72 20 4f 2c 6f 5b 5d 3b 6d 61 69 6e 28 char O,o[];main(
6c 29 7b 66 6f 72 28 3b 7e 6c 3b 4f 7c 70 75 1){for(;~1;O||pu
74 73 28 6f 29 29 4f 3d 28 4f 5b 6f 5d 3d 7e 28 ts(o))O=(O[o]=~(
6c 3d 67 65 74 63 68 61 72 28 29 29 3f 34 3c 28 l=getchar()))?4<(
34 5e 6c 3e 3e 35 29 3f 6c 3a 34 36 3a 30 29 3f 4^1>>5)?l:46:0)?
2d 7e 4f 26 70 72 69 6e 74 66 28 22 25 30 32 78 ~O&printf("%02x
20 22 2c 6c 29 2a 35 3a 21 4f 3b 7d 0a ",l)*5:!O;}.

```

2014: Yusuke Endoh, double helix synthesiser

The following program *prog.c* is formatted as a double helix:

切换行号显示

```
1 #include <stdio.h>
2 #define TA q=/*XYXY*/
3 #define/*X YXY*/CG r=
4 void p(int n,int c){;
5 for(;n--;) putchar(c)
6 #define Y(z)d;d=c++\
7 %2<1?x=x*4 +z,c%8>5?\
8 x=x?p(1,x), 0:x:0:0;d=
9 #define/*X YX*/C Y(1)
10 #define/*X YX*/G Y(2)
11 ;}int(*f)( void),d,c,
12 #define/*X YX*/A Y(0)
13 #define/*XY*/AT int\
14 m(void/**/){d=
15 #define/*XYX*/T Y(3)
16 #define GC d; return\
17 0;}int(*f) (void )=m;
18 x,q,r; int main(){if(
19 f)f();else {for(puts(
20 "#include" "\40\"pro\
21 g.c\"\\n\\n \\101T"+0);
22 d=!d?x=(x= getchar())
23 <0?0:x,8*8 :d,TA++c%8
24 ,TA(1+7*q- q*q)/3,r=c
25 *15-c*c-36 ,p(r<0?!q+
26 4:r/6+!q+4 ,32),q||x;
27 c%=16)q?p( 1,"ACGT"[x
28 /d&3]),p(q ,126),p(1,
29 "TGCA"[x/d &3]),d/=4,
30 p(001,10): puts(c%8?\
```

```
31 "CG":"TA")    ;puts("GC"
32 );}return    0;}/**/
```

Reminder: the *echo* command just sends the string arguments to *stdout* (i.e. the screen here)

```
prompt$ echo love is blind
love is blind
```

Compiling, and executing Yusuke Endoh's program with input string 'owl':

```
prompt$ gcc prog.c
prompt$ echo owl | ./a.out
#include "prog.c"
```

```
AT
C~~G
G~~~C
T~~~~A
T~~~~A
C~~~G
T~~A
CG
TA
C~~G
T~~~A
C~~~~G
G~~~~C
T~~~A
A~~T
CG
TA
A~~T
A~~~T
G~~~~C
G~~~~C
A~~~T
A~~T
GC
```

That output is a C program, believe it or not!, that can also be compiled and executed:

```
prompt$ gcc prog.c
prompt$ echo owl | ./a.out > owl.c
prompt$ gcc owl.c
prompt$ ./a.out
owl
```

So, we've come full circle:

- the output, when compiled (!), generates new output that is the same as the original input

20??: ???

Can you see what the following program does?

切换行号显示

```
1 m(f,a,s)char*s;
2 {char c;return f&1?a!=*s++?m(f,a,s):s[11]:f&2?a!=*s++?
1+m(f,a,s):1:f&4?a--?
3 putchar(*s),m(f,a,s):a:f&8?*s?m(8,32,(c=m(1,*s++,"Arjan Kenter.
\no$../.\\"),
```

```

4
m(4,m(2,*s++,"POCnWAUvBVxRsoqatKJurgXYyDQbzhLwkNjdMTGeIScHFmplizEf"),&c
),s)):
5 65:
(m(8,34,"rgeQjPruaOnDaPeWrAaPnPnPrCnOrPaPnPjPrCaPrPnPrPaOrvaPndeOrAnOrPnO
rP\
6
nOaPnPjPaOrPnPnPrPtPnPnPrAaPnBrnnsrnnBaPeOrCnPrOnCaPnOaPnPjPtPnAaPnPnPrPn
PrCaPn\
7
BrAnxrAnVePrCnBjPrOnvrCnxrAnxrAnsrOnvjPrOnUrOnornnsrnnorOtCnCjPrCtPnCrn
nirWtP\
8
nCjPrCaPnOtPrCnErAnOjPrOnvtPnnrCnNrnnRePjPrPtNrUnnrntPnbtPrAaPnCrnnOrPj
PrRtPn\
9
CaPrWtCnKtPnOtPrBnCjPronCaPrVtPnOtOnAtnrxaPnCjPrqnnPrtaOrsaPnCtPjPratP
nnaPrA\
10
aPnAaPtPnnaPrvaPnnjPrKtPnWaOrWtOnnaPnWaPrCaPnntOjPrRtOnWanrOtPnCaPnBtCj
PrYtOn\
11 UaOrPnVjPrwtnnxjPrMnBjPrTnUjP"),0);}
12 main(){return
m(0,75,"mIWltouQJGsBniKYvTxODafbUcFzSpMwNCHEgrdLaPkyVRjXeqZh");}

```

Compiling using gcc generates lots of warnings

Execution generates output

Assignments and conditions are expressions

Assignment statements are really expressions:

- they return a result: the value being assigned
 - usually the return value of an assignment statement is *thrown away*

Assignment statements can be used in conditions.

- instead of testing a 'Boolean', test the returned value from an assignment statement

Example:

- assuming *getNext()* returns the next 'whatever', and 0 when there is no more
- the fragment

切换行号显示

```

1 v = getNext();
2 while (v != 0) {
3     ...           // do something with v
4     v = getNext();
5 }

```

- can better be written as

切换行号显示

```

1 while ((v = getNext()) != 0) {
2     ...           // do something with v
3 }

```

Conditionals are normally used in if-statements, if-else-statements and while-loops

切换行号显示

```
1 if (x >= 0) {
2     printf("x = %d\n", x);
3 }
4
while (x >= 0) {
    printf("x = %d\n", x);
    ...
}
```

but they can also appear in assignment statements

- *What is the final value of x in the 3 code fragments?*
 - In UNIX, *false* has the value 0, and *true* the value 1

切换行号显示

```
1 x = 42;
2 x = (x>=0) + (x<0);
+ (x<0);
x = -42
x = (x>=0) + (x<0);
x = 0;
x = (x>=0)
```

- the final value of x in all 3 fragments is 1
- *What about?*

切换行号显示

```
1 x = 42;
2 x = (x>0) + (x<0);
(x<0);
x = -42
x = (x>0) + (x<0);
x = 0;
x = (x>0) +
```

- the final values of x are 1, 1 and 0 respectively

Types

C is a typed language, so every variable must have a (fixed) type

The main basic types are:

- char character 'A', 'e', '#', ...
- int integer 2, 17, -5, ...
- float floating-point number 3.14159, ...
- double double precision floating-point 3.14159265358979, ...

From these basic types, more complex types can be created

- arrays: a homogenous aggregate data type
- structs: a heterogeneous aggregate data type

Arrays

An array is

- a linear sequence of same-type variables
- accessed using an integer index
- for an array of size N , valid indexes are $0..N-1$

切换行号显示

```
1 #define MAX 20
2
3 ...
4
5 int i; // index used in the array
```

```

6     int fac[MAX];    // array of MAX 'int' values
7
8     fac[0] = 1;
9     for (i = 1; i < MAX; i++) {
10         fac[i] = i * fac[i-1];
11     }

```

Array initialisation

Can initialise arrays when you declare them:

切换行号显示

```

1 // arrayint.c
2 #define LENGTH 5
3
4 #include <stdio.h>
5 int main()
6 {
7     int v[LENGTH] = {1,2,3,4,5};
8
9     printf("v = ");
10    for (int i=0; i<LENGTH; i++) {
11        printf ("%d ", v[i]);
12    }
13    putchar('\n');
14    return 0;
15 }

```

Output is:

```
v = 1 2 3 4 5
```

Change the length, *#define LENGTH 20* generates:

```
v = 1 2 3 4 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Arrays of 'char' are referred to as strings and can be printed as a string.

切换行号显示

```

1 // arraychar.c
2 #include <stdio.h>
3 int main(void)
4 {
5     char s[6] = {'h','e','l','l','o'}; // room for a null
terminator
6     char t[6] = "hello";              // room for a null
terminator
7
8     printf ("s = %s and t = %s\n", s, t);
9     return 0;
10 }

```

The two char arrays are equivalent (almost). Output is:

```
s = hello and t = hello
```

Notice the arrays are bigger than the string length.

What happens if the array is just long enough (no room for the null)?

1. `s[5]`? Bad news, at runtime buffer overflow error
2. `t[5]`? Same buffer overflow error.

What happens if you do not specify the array length at all?

1. `s[]`? Still a buffer overflow error.
2. `t[]`? This works fine.

Array passing to functions

Passing an array to a function involves passing the address of the first element of the array

- this is very efficient
- but the function does not know the length of the array

Need to also pass the length to the function

- so a function could look like:

切换行号显示

```
1  int sumArray(int harray[], int length) {
2      int total = 0;
3      for (int i=0; i<length; i++) {
4          total += harray[i]; // could be written total = total
+ harray[i]
5      }
6      return total;
7  }
```

An example of a call to the function as well

切换行号显示

```
1  int v[] = {1,2,3,4,5};
2  printf ("Sum of array is %d\n", sumArray(v, 5));
```

Notice the argument `v` in the call statement `sumArray(v, 5)`.

- that is just an address of where `v` starts (i.e. the address of `v[0]`)

You could modify `sumArray()` to treat `harray` as an address, also called a pointer

- and treat `i` as an *increment* rather than an index

切换行号显示

```
1  int sumArray(int *harray, int length) { // this says harray
is a pointer to some data
2      int total = 0;
3      for (int i=0; i<length; i++) {
4          total += *(harray + i); // you could still have
used harray[i] here
5      }
6      return total;
7  }
```

- this does the same thing as before
- note we haven't changed the main function at all

The function *sumArray()* is an example of using 'pointer arithmetic' to process data

Arrays of multi-dimensional

A two-dimensional array is an array of arrays:

- so $m[2][3]$ is an array of 2 elements, each of which is an array of size 3 elements
 - so $m[0]$ consists of the array elements $m[0][0]$, $m[0][1]$ and $m[0][2]$
 - that is, the first row
 - and $m[1]$ consists of the array elements $m[1][0]$, $m[1][1]$ and $m[1][2]$
 - that is, the second row

For example, the array

```
1.1  2.2  3.3
4.4  5.5  6.6
```

is represented by:

切换行号显示

```
1 int m[2][3] = {{1.1, 2.2, 3.3},
2               {4.4, 5.5, 6.6}
3               };
```

A three-dimensional array example:

- $m[3][2][4]$ is an array of 3 elements, where each element is an array with 2 rows x 4 columns

切换行号显示

```
1 int m[3][2][4] = {{{1, 2, 3, 4}, {5, 6, 7, 8}},           // row 0,
consisting of 2 rows of 4
2                  {{9, 10, 11, 12}, {13, 14, 15, 16}},    // row 1,
consisting of 2 rows of 4
3                  {{17, 18, 19, 20}, {21, 22, 23, 24}}    // row 2,
consisting of 2 rows of 4
4                  };
```

Example, summing a $[2][3]$ array of floats:

切换行号显示

```
1 // arraymulti.c
2
3 #define NROWS 2
4 #define NCOLS 3
5
6 #include <stdio.h>
7 int main(void) {
8     float m[NROWS][NCOLS] = {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}};
9     float totalxR = 0, totalxC = 0;
10
11     //sum row-by-row
12     for (int row = 0; row < NROWS; row++) {
13         for (int col = 0; col < NCOLS; col++) {
14             totalxR += m[row][col];
15         }
16     }
```

```

17 // alternatively sum column-by-column
18 for (int col = 0; col < NCOLS; col++) {
19     for (int row = 0; row < NROWS; row++) {
20         totalxC += m[row][col];
21     }
22 }
23 printf ("totalxR = %lf and totalxC = %lf\n", totalxR,
totalxC);
24 }

```

Output

```
totalxR = 23.100000 and totalxC = 23.100000
```

Multi-dimension array passing to functions

If you *#define* the size of the 2D array ...

- then an example of a function that prints a 2D array is

切换行号显示

```

1 void printf2 (float harry[NROWS][NCOLS]) {
2     for (int i = 0; i < NROWS; i++) {
3         for (int j = 0; j < NCOLS; j++) {
4             printf("%f ", harry[i][j]);
5         }
6         putchar('\n');
7     }
8 }

```

If we take the previous program, the call to print the array is simply:

切换行号显示

```
1 printf2(m);
```

Structs

A *struct*

- is a collection of variables, often of different types, grouped together in a single entity
- helps manage complexity of data
- enforces links/relationships between data
- example:

切换行号显示

```

1 struct date {
2     int day;
3     int month;
4     int year;
5 };

```

A *struct* definition just specifies a type

- to allocate memory you need to define a variable

切换行号显示

```
1 struct date birthday;
```

Typedefs

C allows the user to define a new type name (in terms of existing types of course)

- if you don't like the name *float*, and wish to use *Real* ...

切换行号显示

```
1 typedef float Real;
2 ...
3 Real getAverage(Real a, Real b) { // a simple function
4     return (a+b)/2;
5 }
```

More seriously, a *typedef* is often used for a *struct* to clean up code

切换行号显示

```
1 typedef struct {
2     int a;
3     int b;
4 } Pair;
5 ...
6 Pair x;
```

You can nest and combine types, normal C types or those created by you, in structs:

切换行号显示

```
1 typedef struct {
2     int day, month, year;
3 } Date;
4 typedef struct {
5     int hour, minute;
6 } Time;
7 typedef struct {
8     char plate[7]; // e.g. "ABC123D"
9     float speed;
10    Date d;
11    Time t;
12 } SpeedingTicket;
```

In the program that accesses this data type, you do the following:

切换行号显示

```
1 #define NUM_TICKETS 10000
2
3 typedef struct {
4     ... // all the above
5 } SpeedingTicket;
6
7 SpeedingTicket tickets[NUM_TICKETS]; // an array of structs
8
9 // Print all speeding tickets in a readable format
10 for (int i = 0; i < NUM_TICKETS; i++) {
11     printf("%s %6.3f %d-%d-%d at %d:%d\n", tickets[i].plate,
12         tickets[i].speed,
13         tickets[i].d.day,
14         tickets[i].d.month,
15         tickets[i].d.year,
16         tickets[i].t.hour,
```

```

17                                     tickets[i].t.minute);
18 }

```

Accessing data in structs

The following program shows how to use a *struct*, and how to not use *struct*

切换行号显示

```

1 // pointerstruct.c
2 #include <stdio.h>
3
4 typedef struct {
5     int name;
6     int salary;
7 } Worker;                                // Worker is now a type, like char
and int
8
9 int main(void) {
10     Worker w, *wp;                       // define variables with the new type
11
12     wp = &w;
13     *wp.salary = 125000;                 // generates a compiler error (see
below)
14     *(wp.salary) = 125000;               // so does this (same as previous
line)
15
16     w.salary = 125000;                   // is probably what you want
17     (*wp).salary = 125000;               // or you can use the pointer, but
this looks kludgy
18     wp->salary = 125000;                 // this is the 'sexy' way to use
pointers
19
20     return 0;                            // being lazy, should be EXIT_SUCCESS
21 }

```

when compiled and executed will result in:

```

pointerstruct.c:11: error: member reference type 'Worker *' is a pointer;
did you mean to use '->'?
pointerstruct.c:12: error: member reference type 'Worker *' is a pointer;
did you mean to use '->'?

```

In general: if you have:

切换行号显示

```

1 structTypeName  s, *sp;
2 sp = &s;

```

then the following are equivalent:

```

s.element
(*sp).element
sp->element

```

Command-Line Arguments

All the executable programs above have a *main(void)* program

- more generally, executables take arguments on the command line
- these enter the program via parameters

切换行号显示

```
1  int main(int argc, char *argv[])
```

For example:

```
prompt$ ./a.out -pre 3
```

means that:

- *argc* refers to the number of arguments, including the program name
 - *argc* = 3
- *argv[]* allows access to arguments represented as strings
 - *argv[0]* is a pointer to the string `./a.out`
 - *argv[1]* is a pointer to the string `-pre`
 - *argv[2]* is a pointer to the string `3`

Command-line argument processing example 1

Write a program that:

- takes an optional numerical command-line argument 1, 2, 3 or 4
- if the argument is not one of these characters (!), a usage message is printed

切换行号显示

```
1  // commarg.c
2  #include <stdio.h>
3  #include <stdlib.h>
4  int main(int argc, char *argv[]) {
5      if (argc == 1 ||
6          (argc == 2 && atoi(argv[1]) >= 1 && atoi(argv[1]) <=
7  4)) {
8          // we can do something here
9      } else {
10         printf("Usage: %s [1|2|3|4]\n", argv[0]);
11     }
12     return EXIT_SUCCESS;
13 }
```

- notice that *atoi()* had to be called to convert the character to a number
- compiling and executing the program:

```
prompt$ gcc commarg.c
prompt$ ./a.out
prompt$ ./a.out 1
prompt$ ./a.out 2
prompt$ ./a.out 3
prompt$ ./a.out 4
prompt$ ./a.out 5
Usage: ./a.out [1|2|3|4]
```

Command-line argument processing example 2

Write a program that:

- takes an optional command-line switch *-reverse*
- if the switch is not correct, a usage message is printed

切换行号显示

```

1  // commargrev.c
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  int main(int argc, char *argv[]) {
6      if (argc == 1 ||
7          (argc == 2 && !strcmp(argv[1], "-reverse"))) {
8          // NOTE: strcmp returns 0 if matches.
9          // we could do something here
10     } else {
11         printf("Usage: %s [-reverse]\n", argv[0]);
12     }
13     return EXIT_SUCCESS;
14 }
```

```

prompt$ ./a.out
prompt$ ./a.out -reverse
prompt$ ./a.out rubbish
Usage: ./a.out [-reverse]
```

Makefile

There is a utility in UNIX called *make* that can save you time and help you organise your files. The utility executes the commands in a file called *Makefile*. A *Makefile* consists of *rules*. An example of a rule is the following:

```

definitions

target: dependencies
      commands
```

- the *definitions* define variables. It is optional.
- the *target* is the name of the object (usually an executable) you wish to make
 - you may have multiple targets (see below)
- the *dependencies* is a list of files that the object depends on
 - this is usually a list of C files and header files
- the *commands* is a list of instructions that will make the *target*
 - this is usually a compilation
 - each command should be preceded by a tab

An example of a rule in a *Makefile* is:

```

ex1: ex1.c ex1.h
    dcc -O3 -o ex1 ex1.c
```

It is important to note that *dcc* above is preceded by a tab (and not spaces).

Every file in UNIX has a *timestamp*, which is the time it was created or last modified.

- If you type in the command *make*, then *make* will compare the timestamps of each of the files in the dependencies with the target.

- If any of these dependency files is newer than the target, *make* will execute the command again (which usually makes the target again).
- If none of the files is newer, *make* simply reports that the target is up to date.

You can have multiple targets. Below we show three targets:

```
all: ex1 ex2

ex1: ex1.c ex1.h
    dcc -O3 -o ex1 ex1.c

ex2: ex2.c ex2.h
    dcc -O3 -o ex2 ex2.c
```

You can then type *make ex1* or *make ex2* as you wish.

If you type simply *make*, then the default is to make the first target only, which in this case is *all*, which in turn has as its dependencies the other two targets. This command will make either or both the executables if they are older than their course code.

You can generalise the *Makefile* by defining and using variables, such as:

```
CC=dcc
CFLAGS=-O3

all: ex1 ex2

ex1: ex1.c ex1.h
    $(CC) $(CFLAGS) -o ex1 ex1.c

ex2: ex2.c ex2.h
    $(CC) $(CFLAGS) -o ex2 ex2.c
```

The variables will be substituted when the *Makefile* is executed.

You can also add a target to clean up your directory after you have finished:

```
CC=dcc
CFLAGS=-O3

all: ex1 ex2

ex1: ex1.c ex1.h
    $(CC) $(CFLAGS) -o ex1 ex1.c

ex2: ex2.c ex2.h
    $(CC) $(CFLAGS) -o ex2 ex2.c

clean:
    rm -f ex1 ex2 ex1.o ex2.o core
```

- the *clean* target has no dependencies, and here removes the executable, the object files, and core dumps
- the command *make clean* will execute the *rm* instruction
- the *-f* switch means that *rm* will not complain if any files are do not existent

special macros

Special macros, or abbreviations, are predefined in *make*. Two examples are:

- `$$` which is an abbreviation of the target name
- `$(CC)` which is an abbreviation of the source filename corresponding to the target

An example of their use:

```
prog: prog.c
      $(CC) $(CFLAGS) -o $$ $<
```

touch the timestamp

The UNIX command **touch** can be used to make the timestamp of a file the current time. For example:

```
touch prog1.c
```

will change the timestamp of *prog1.c* to the current time. This is a useful command to 'force' *make* to re-make an executable, for example.

Asserts

assert() is a function that can be called from anywhere within your program

- *need to include the file `assert.h`*

The manual entry for *assert* (i.e. type in *man assert*) generates the following:

```
ASSERT(3)                Linux Programmer's Manual
ASSERT(3)

NAME
    assert - abort the program if assertion is false

SYNOPSIS
    #include <assert.h>

    void assert(scalar expression);

DESCRIPTION
    If the macro NDEBUG was defined at the moment <assert.h> was
    last included, the macro assert() generates no code, and hence
    does nothing at all. Otherwise, the macro assert() prints an error
    message to standard error and terminates the program by calling abort(3) if
    expression is false (i.e., compares equal to zero).

    The purpose of this macro is to help the programmer find bugs in
    his program. The message "assertion failed in file foo.c, function
    do_bar(), line 1287" is of no help at all to a user.

RETURN VALUE
    No value is returned.

CONFORMING TO
    POSIX.1-2001, C89, C99. In C89, expression is required to be of
    type int and undefined behavior results if it is not, but in C99
    it may have any scalar type.

BUGS
    assert() is implemented as a macro; if the expression tested has
```

side-effects, program behavior will be different depending on whether NDEBUG is defined.

Does assert help handle errors?

Here is an example of an assert used to catch a potential error.

切换行号显示

```

1 // divideDebug.c: divide the first command-line argument by the
second
2 // There must be 2 arguments on the command line!
3 // Program assumes that any arguments on the command line are
numerical
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <assert.h>
7
8 int main(int argc, char *argv[]) {
9     int n, d;
10    if (argc == 3) {
11        sscanf(argv[1], "%d", &n); // for readability leave out
check okay
12        sscanf(argv[2], "%d", &d); // for readability leave out
check okay
13        assert(d!=0);
14        printf("%f\n", 1.0*n/d);
15    }
16    else {
17        printf("Usage: %s num den\n", argv[0]);
18    }
19    return EXIT_SUCCESS;
20 }

```

Compile the program and execute with a 0 denominator:

```

prompt$ gcc divideDebug.c
prompt$ ./a.out 1 0
a.out: divideDebug.c:12: main: Assertion `d!=0' failed.
Aborted

```

The assert is telling us:

- name of the executable (*a.out*)
- name and line number of the source file (*divideDebug.c:12*)
- name of the function (*main*)
- why assert has been triggered (the argument *d!=0* is false)
- finally, it has aborted the program (self-destruct)

This information is useful to a programmer, as debug. It is not useful to a user. An assert:

- crashes the program
- should only be used to debug
- should never appear in 'production' code
 - *what good is a crash to an end-user, what is he supposed to do?*

In program design, *exceptions* are 'caught'. An *exception*:

- is an error that is handled by the program
- often dealt with by an error handler

- the resulting error message should be informative
- at worst, the error handler should terminate the program gracefully

Can we rewrite the *divideDebug* program above to be more graceful?

切换行号显示

```

1 // divide.c: divide the first command-line argument by the second
2 // There must be 2 arguments on the command line.
3 // Program assumes that any arguments on the command line are
numerical
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main(int argc, char *argv[]) {
8     int n, d;
9     if (argc == 3) {
10         sscanf(argv[1], "%d", &n); // for readability leave out check
okay
11         sscanf(argv[2], "%d", &d); // for readability leave out check
okay
12         if (d!=0) {
13             printf("%f\n", 1.0*n/d);
14         }
15         else {
16             fprintf(stderr, "Error: the denominator cannot be
zero\n");
17             return EXIT_FAILURE; // this is terminating gracefully
18         }
19     }
20     else {
21         printf("Usage: %s num den\n", argv[0]);
22     }
23     return EXIT_SUCCESS;
24 }

```

Note:

- the value of *EXIT_SUCCESS* is 0, *EXIT_FAILURE* is 1

To see 'error handling' in action, type in a 0 denominator:

```

prompt$ ./a.out 1 0
Error: the denominator cannot be zero

```

```

prompt$ ./a.out 1 2
0.500000

```

NDEBUG and assert

What's this NDEBUG business in the assert man entry?

- it is used to turn the *assert* on and off
 - if defined it says 'no debug', which turns the *assert* off
 - if not defined the *assert* will be turned on

Consider the following code:

切换行号显示

```

1 // echoArg.c
2 // a really bad program to print the argument on the command line
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <assert.h>
6 int main(int argc, char *argv[]) {
7     assert(argc==2);
8     printf("%s\n", argv[1]);
9     return EXIT_SUCCESS;
10 }

```

Below I compile, and execute, *echoArg.c* with 1) **NDEBUG** defined and 2) **NDEBUG** not defined

1. define **NDEBUG**

- do this by using the *-D* option

```

prompt$ gcc -D NDEBUG echoArg.c
prompt$ ./a.out
(null)

```

- *argv[1]* does not exist so *printf()* is printing *nothing*
- clearly a runtime error
 - above *gcc* was used: if you use *gcc* the program generates a segmentation violation

2. do not define **NDEBUG**

- do this by just compiling normally

```

prompt$ gcc echoArg.c
prompt$ ./a.out
a.out: echoArg.c:7: int main(int, char **): Assertion
`argc==2' failed.

Execution stopped here in main() in echoArg.c at line 7:

#include <assert.h>
int main(int argc, char *argv[]) {
--> assert(argc==2);
    printf("%s\n", argv[1]);
    return EXIT_SUCCESS;
when execution stopped:
    argc = 1
    argv[1] = NULL

```

- in effect, the *assert* has crashed the program

Use *CFLAGS* in your *Makefile* to select/deselect the *NDEBUG* option.

If you submit a *Makefile* in this course, and you use asserts, make sure you include *NDEBUG* so all asserts are turned off.

A better version of the program is the following:

切换行号显示

```

1 // echoArgBetter.c
2 // The program echoArg.c done better
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main(int argc, char *argv[]) {

```

```
7     if (argc != 2) {
8         fprintf(stderr, "Usage: %s arg\n", argv[0]);
9     }
10    else {
11        printf("%s\n", argv[1]);
12    }
13    return EXIT_SUCCESS;
14 }
```

- no assert in this program
- the number of arguments on the command line is 'caught'
 - an appropriate error message is printed

```
prompt$ ./a.out
Usage: ./a.out arg
```

```
prompt$ ./a.out abc
abc
```

Lec01Revision (2019-06-16 12:44:25由AlbertNymeyer编辑)