RMIT UNIVERSITY
Computer Science and Information Technology

Advanced Programming Techniques

(a.k.a. Programming in ANSI / ISO C)

Module 10 – Readability, Usability, Command-line arguments, Unions

"Shaw's Principle: Build a system that even a fool can use,
and only a fool will want to use it."

Readability

- Readability of source code: a somewhat subjective assessment of how easily an experienced programmer can read and understand source code.

- High readability can assist with debugging as well as future maintenance of code. Obviously important in a team situation.

Obfuscated C Code

An extreme, and amusing, example of an actual C program with very low readability

```
                                                                      char
                                                                    _3141592654[3141
          ],__3141[3141];_314159[31415],_3141[31415];main(){register char*
        _3_141,*_3_1415, *_3__1415; register int _314,_31415,__31415,*_31,
      _3_14159,__3_1415;*_3141592654=__31415=2,_3141592654[0][_3141592654
     -1]=1[__3141]=5;__3_1415=1;do{_3_14159=_314=0,__31415++;for( _31415
    =0;_31415<(3,14-4)*__31415;_31415++)_31415[_3141]=_314159[_31415]= -
  1;_3141[*_314159=_3_14159]=_314;_3_141=_3141592654+__3_1415;_3_1415=
  __3_1415    +__3141;for                 (_31415 = 3141-
             __3_1415  ;                  _31415;_31415--
             ,_3_141 ++,                  _3_1415++){_314
             +=_314<<2 ;                  _314<<=1;_314+=
            *_3_1415;_31                   =_314159+_314;
            if(!(*_31+1)                   )* _31 =_314 /
            __31415,_314                   [_3141]=_314 %
            __31415 ;* (                   _3__1415=_3_141
           )+= *_3_1415                     = *_31;while(*
           _3__1415 >=                      31415/3141 ) *
           _3__1415+= -                     10,(*--_3__1415
          )++;_314=_314                     [_3141]; if ( !
          _3_14159 && *                     _3_1415)_3_14159
          =1,__3_1415 =                     3141-_31415;}if(
          _314+(__31415                      >>1)>=__31415 )
          while ( ++ *                       _3_141==3141/314
         )*_3_141--=0                        ;}while(_3_14159
         ) ; { char *                        __3_14= "3.1415";
         write((3,1),                        (--*__3_14,__3_14
         ),(_3_14159                          ++,++_3_14159))+
        3.1415926; }                          for ( _31415 = 1;
       _31415<3141-                           1;_31415++)write(
      31415% 314-(                            3,14),_3141592654[
    _31415    ] +                            "0123456789","314"
    [ 3]+1)-_314;                           puts((*_3141592654=0
  ,_3141592654))                             ;_314= *"3.141592";}       /* http://www.ioccc.org/  */
```

Readability

- Many organisations have ‘coding standards’, enabling a common ‘style’ to be used across a programming team. There are software tools to help check code adheres to specified standards.

- Common topics covered by a coding standard include:
- Format of symbolic constants
  - E.g. all uppercase, nouns only - no verbs, maximum length allowed etc

- Format of other identifiers
  - E.g. camelCase notation (each word except first is capitalised e.g. ‘maxAge’), variables are often nouns, functions are often verbs etc

Readability

- Code layout
  - E.g. no line > 80 characters, indented 1 tab character for each block level, open/close braces occur on new lines, double newline between functions, maximum permitted level of nesting etc
- File organisation:
  - E.g. Code divided into modules, each of which performs a logically separate functionality and having a clearly specified interface. A particular style of folder organisation may be specified
- Documentation:
  - Within source files: e.g. describe the intent of each function and it's parameters and any 'difficult' code. Describe 'why' instead of 'what', avoiding the obvious.
  - External documentation standards

Usability

- Usability relates to the relationship between software and the users of the software. It is the quality of a system that makes it easy to learn, easy to use, easy to remember, error tolerant, and subjectively pleasing.

- Given a choice, most people will tend to buy systems that are more user-friendly. We can learn to be better user interface designers by learning design principles and design guidelines.

- The study of usability is a course (or two) unto itself and well outside the scope of this course.

User-interfaces

- User-interfaces are of course the aspects of a computer system or program which can be seen (or heard or otherwise perceived) by the human user, and the commands and mechanisms the user uses to control its operation and input data.

- There are a wide range of user-interface options, including, but not limited to:
  - Text-based menu-driven
  - Text-based command line interface (CLIs)
  - Text-based/Window user prompts / user input
  - Graphical user interfaces (GUIs)

User-interfaces (cont'd)

- ANSI C provides no specific language support for any but the most basic of text-based user interfaces, however many third-party libraries exist to provide support for sophisticated user-interfaces.

- For portability, it is often best for user-interfaces to be well isolated from the essential application, by way of functional/modular abstraction (ie. abstraction layer).

- C does provide limited support for command line arguments to be accepted by the program at the time the program is invoked by the operating system.

Command line arguments

- Used to pass data to function `main()` when you invoke the program from the command line - e.g.
  - if the program on the following slide is compiled into an executable file called '`add`'
  - then entering the following command at the (Unix) prompt

```
./add  -123  45  777
```

would produce output as shown:

```
output 699
```

Command line arguments (cont'd)

```
int main(int argc, char *argv[])
{
  int sum = 0, i;
  for (i=1; i<argc; i++)
     sum += atol(argv[i]);
  printf("output %d\n", sum);
  return EXIT_SUCCESS;
}
```

why initialise i=1 instead of 0?

Unions

- *Union* defines a type (but not storage) similarly to *struct*
- Unlike struct, members share memory (they 'overlap' in memory)
  - Enough storage is allocated to hold the largest member of the union

```
union IDcode {
        int numID;
        char nameID[10];
};

union IDcode myID;
```

Unions

```
union IDcode {
        int numID;
        char nameID[10];
};

union IDcode myID;
myID.numID = 123; /* interpret memory as int */
strcpy(myID.nameID,"Lin");
                /* memory storing int 123 now
                   overwritten by string */
```

- How the memory is interpreted depends on which member name the programmer uses.

Unions

- Assume any vehicle is either a car or a bus (but not both)

```
typedef struct {
    int  numCylinders;
    int  capacity;
} CarType;
```

```
typedef struct  {
    int maxPass;
    char class[20];
} BusType;
```

Unions (cont'd)

```
typedef union  {
   CarType  car;
   BusType  bus;
}  BusOrCar;

typedef struct  {
   char  manufacturer[30];
   int   engineNum;
   char  vehicleCode; /* 'C'=car, 'B'=bus  */
   BusOrCar  other;
}  VehicleType;
```

Unions (cont'd)

```
VehicleType  v1, v2;

strcpy(v1.manufacturer, "Holden");
v1.engineNum = 12345;
v1.vehicleCode = 'C';
v1.other.car.numCylinders = 6;
v1.other.car.capacity = 2800;

strcpy(v2.manufacturer, "Volvo");
v2.engineNum = 98765;
v2.vehicleCode = 'B';
v2.other.bus.maxPass = 48;
strcpy(v2.other.bus.class, "Commercial");
```