UPPSALA UNIVERSITET

Methods of Programming DV2

Coding and documentation standard

General

- The coding standard is not absolute. The important thing is not that it is followed to the letter, but in spirit.
- Be clear

  By clearly expressing what is to be done, the code will be easier to understand and modify.

  Don't use "clever tricks" which achieves short and/or more efficient code by obscure means.

  Don't write code that works "by accident". That is, the code is wrong in general, but happens to give the right result for the particular case(s) where it is used.
- Be consistent

  Being consistent - in constructing, formatting as well as documenting code - increases readability and maintainability. Consistency should apply also between members of each project group.
- Be abstract

  Abstract code focuses on the task to be done rather than how it is done. It is substantially easier to maintain abstract code as changes in general are kept within limited parts of the code.

Documentation (1)

- All code should be documented.

  The code in itself expresses the concrete way things are done. Someone reading the code is more interested in an abstract view.
- Code should be implicitly documented

  Implicit documentation in the form of meaningful function and variable names enhance readability

Documentation (2)

- Code should be explicitly documented

  Explicit documentation in the form of function headers and comments enhance readability and understanding. Note that there are different prospective readers of explicit documentation. Someone who wants to change something needs more concrete descriptions while someone who wants to use the code (as a library) wants something more abstract.

```
/*-------------------------------------------------------------*
 * Function     : List_Insert(List l, ElementRef e) -> void    *
 * Input        : l     Any list.                              *
 *                e     Element to insert into list.           *
 * Precondition : none                                         *
 * Return       : void                                         *
 * Effects      : Destructively inserts element e somewhere into *
 *                list l.                                      *
 * Ownership    : l declares ownership of e.                   *
 *-------------------------------------------------------------*/
  void List_Insert(List l, ElementRef e) {
      ...
  }
```


Documentation (3)

- Code should be accompanied by external documentation

  Dependencies between different files/modules are best expressed using documents separate from the code. The same goes for installation instructions and user guides. As for other kinds of explicit documentation, there can be several different audiences, with different demands on how the documentation is written.

Modules

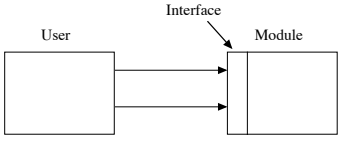
A module is a part of the program (set of procedures) that provide a related service to other parts of the program (users).

The services are made available through the *interface* of the module. The interface is simply the names and types of the procedures and variables of the module which are intended to be accessed by the user.

Other (internal) functions of the module are *hidden* from the user.

An important use of modules is to implement abstract data types.

Many languages support the module concept directly, but C does not.



Modules in C

Modules i C can be simulated by keeping the interface in a `.h` file and the implementation in a corresponding `.c` file. A `.h` file may only include declarations. A `.c` file can have both declarations and definitions.

`vector.c:`

```
#include "vector.h"
…
Vector vector_create(float x,
                     float y,
                     float z){
   …
};

…
static float aux(float ff){
   …
};
…
```

`vector.h`

```
…
typedef struct {
    float x,y,z;
} Vector;

Vector vector_create(float x,
                     float y,
                     float z);
…
```

Other modules that use vectors include the file `vector.h`.
The internal function `aux` is not accessible outside the vector module.

Modules guidelines (1)

- Export as little information as possible

  Every name which is visible outside of a module is a potential for name collision. Visible information might lure another user into writing representation-dependent code. Use `static` on procedure and (global) variable names that should not be used outside the module.

- Use consistent naming principles within every module

  The chance of name collisions is lower and clarity is improved. Additionally, the name itself will be part of the implicit documentation.

- Let the name of the module be a part of exported names

  Same reason as above.

- A module should not define more than one abstract data type

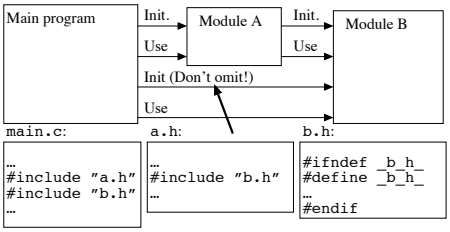
Modules guidelines (2)

- Do not exploit implicit dependencies between modules

  If the implicit dependency is changed, the code will stop working. Also, readibility will suffer.

- Prevent the contents of an `.h` file from being read several times

  Reading the same `.h` file several times can lead to redefinitions of e.g. data types. The code may not compile.



Representation-independent code (1)

- Code should be representation-independent

  Code which is independent of a particular representation code is easier to write, read and maintain.

```
xcoord = v.x;
```

```
xcoord = vector_get_x(v);
```

- Use the same naming convention for computed properties

  The name should not reveal whether a property is stored or computed. If this is changed, the name will imply incorrect information.

```
float vector_compute_length(Vector v){
…
}
```

```
float vector_get_length(Vector v){
…
}
```

- Concrete data should be named

  It is easier to understand a meaningful symbol than a constant value

```
const Vector x_unit_vector = vector_create(1.0,0.0,0.0)
```


Representation-independent code (2)

- Avoid operations which are too specific.

  An(other) user is only interested in the abstract and relevant properties of an abstract data type.
  Example: You have ADTs for (light) Rays and Spheres.
  You want a function to check if a ray will strike a sphere? Where would you put it?
  Not in the Ray ADT (too specific), maybe in the Sphere ADT, but most likely not in an ADT at all

- Abstract away the representation in type definitions

  A type can be simple or composite; a property which should be expressed when it is defined and not when it is used.

```
…
struct Vector {
    float x,y,z;
};

struct Vector
  vector_create(float x,
                float y,
                float z);
…
```

```
…
typedef struct {
    float x,y,z;
} Vector;

Vector vector_create(float x,
                     float y,
                     float z);
…
```


Representation-independent code (3)

- Don't make an object a part of another structure.

  We want to get individual objects that are entirely separate from a containing structure. Also, we want to share objects between structures.

```
struct PersonList_T {
    int          age;
    char         *name;
    PersonList   rest;
};

typedef struct PersonList_T
    *PersonList;
```

```
typedef struct {
    int          age;
    char         *name;
} Person;

struct PersonList_T {
    Person       first;
    PersonList   rest;
};

typedef struct PersonList_T
    *PersonList;
```

- Arrays should be abstract

  An array is a composite data type (size + elements). Also, an array of a particular type can sometimes be represented in a better way than as a straightforward C array. E.g. an array of truth values can be represented as a bit string. Normally use an ADT instead of C arrays.

Representation-independent code (4)

- Truth values must be typed.

Boolean expressions has different semantics compared to expressions which evaluate integers.

Don't do this! Ever!

```
{  int a;

   a = lic(...);
   if (a)
     ...
}
```

Do this if a is really a truth value.

```
typedef enum {
    False = 0,
    True = 1
} Bool;

...

{  Bool a;

   a = lic(..);
   if (a)
     ...
}
```

Do this if a is really an integer or pointer.

```
{ int a;

  a = lic(...);
  if (a != 0)
     ...
}
```

```
{ int *a;

  a = lic(...);
  if (a != NULL)
     ...
}
```

(`lic` = long involved computation)

Macros (1)

- Don't forget that macros use call-by-name

Call-by-name semantics can cause problems.

```
/* Macro to square first argument and assign to y */
#define square(x,y) { float t; t = (x)*(x); y = t; }

  .. foo(t, z) ..                          /* expands to */
  ..  { float t; t = (t)*(t); z = t; } .. /* something undefined */
```


Macros (2)

- Enclose formal parameters in parentheses

Macros use call-by-name at the textual level, so the semantics is not always what you think. If the macro expands to an expression this is true for the entire macro body.

```
#define silly(x,y) x*y

  .. z = silly(a+b, c-d)*t ..    /* expands to */
  .. z = a+b*c-d*t ..            /* != (a+b)*(c-d)*t
*/
```

```
#define silly(x,y) ((x)*(y))

  .. z = silly(a+b, c-d)*t ..    /* expands to */
  .. z = ((a+b)*(c-d))*t ..      /* == (a+b)*(c-d)*t */
```


Variable definitions

- Avoid variable definitions without initial values

You might forget to initialise the variable, leading to undefined behaviour.

Return values

- Avoid returning values using pointer arguments

The code becomes difficult to read since (1) you have to introduce unnecessary temporary variables or (2) it is not obvious that a side-effect it performed on one or more of the arguments. Also, it prevents the use of functional composition.

Memory management (1)

- Stack and heap objects should have different types.

Stack objects are objects local to a procedure or block (storage class `auto`).
Heap objects are objects allocated using `malloc`.
Stack objects has a different (copying) semantics compared to heap objects.

```
typedef struct {
    int    age;
    char   *name;
} Person;

Bool Person_identify(Person *p);
```

```
typedef struct {
    int    age;
    char   *name;
} Person;

Bool Person_identify(Person p);

typedef Person *PersonRef;

Bool PersonRef_identify(PersonRef pref);
```

Memory management (2)

- Never treat a stack object as a heap object (or vice versa)

Don't pass a pointer to a stack object (using `&`). The pointer may still be in use when the object is deallocated. There is also a risk that a heap object will point to the stack. Such a pointer will have a short life-span! An object should keep its particular semantics during its lifetime.

- Always document the intended use of an object

The risk for memory leakage is less if it is clearly stated who will release the memory of the object

```
/*********************************************************
* ADT         : Bitmap                                   *
* Description : ... A bitmap is always heap allocated so it  *
*               only uses reference semantics. ...       *
* Invariant   : The size of the ColourArray should be    *
*               width*height                             *
*********************************************************/
struct Bitmap_T {
    int         width, height;
    ColourArray  pixels;
};
typedef struct Bitmap_T *Bitmap;
```


Error handling (1)

- Error codes should always be handled.

Continued program execution (as if everything was right) is not meaningful if a function returns an error code. It is not that unusual that memory runs out, that the disk becomes full, that a file does not exist etc.

- Always validate user input.

We don't want the program to crash because of bad input data. The user is part of reality and we all know what reality can do to you.

Error handling (2)

- Check that preconditions are satisfied before execution.

Continued program execution (as if everything was right) is not meaningful. We risk undefined behaviour.

```
#include <assert.h>
...
/* True iff list is empty. */
Bool ObjectList_isEmpty(ObjectList ol) {
    return ol == NULL;
};
...
/* Return the first element of a non-empty list. */
Object ObjectList_first(ObjectList ol) {
    assert(!ObjectList_isEmpty(ol));
    return ol->first;
};
```

- Always use sensible error messages.

Facilitates debugging and lets the user know what really went wrong.

```
Bitmap_createFromfile: Couldn't open file 'gazonk.ppm'
```


Flow of control

- Distinguish between expressions and commands

The readability will benefit if you show what you really mean. The most noteworthy difference is that C has both conditional expressions (`?:`) and conditional statements (`if-then-else`). Make a distinction between them and use the right construction at the right time.

- Don't use `goto`

No comment. Really.

Global variables

- Avoid global variables

Global variables cause unnecessary limitations as they only allow one instance of the concept depending on the global variable. Also, it will introduce dependencies between different parts of the code which are difficult to spot.