RMIT UNIVERSITY
Computer Science and Information Technology

Advanced Programming Techniques

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

Module 07 – Module Abstraction and Portability in C

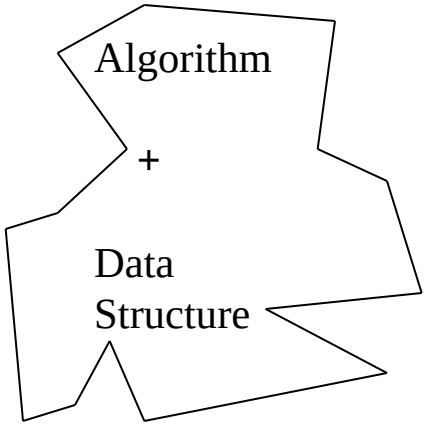
"Dijkstra's Prescription for Programming Inertia:
If you don't know what your program is supposed to do,
you'd better not start writing it."

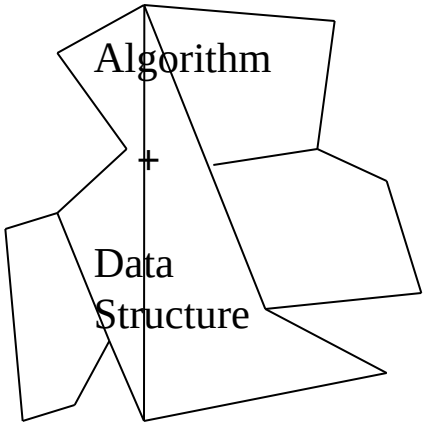
Monolithic Approach

- At its most extreme, this approach consists of one giant function main(). This is clearly not a good thing to do!
- More significantly, the essential algorithm for the application is intermixed with the algorithms for the data structures used by the program.
- Even if multiple functions have been used, the program may still be characteristically monolithic in as much as the essential algorithm for the application and the data structure algorithms are intermixed.
- This makes it difficult, in the extreme, to modify or reuse a data structure.

Monolithic Approach (cont'd)

- Irregular shape implies "messy" solution.
- Even if partitioned into functions, may not be truly modular.

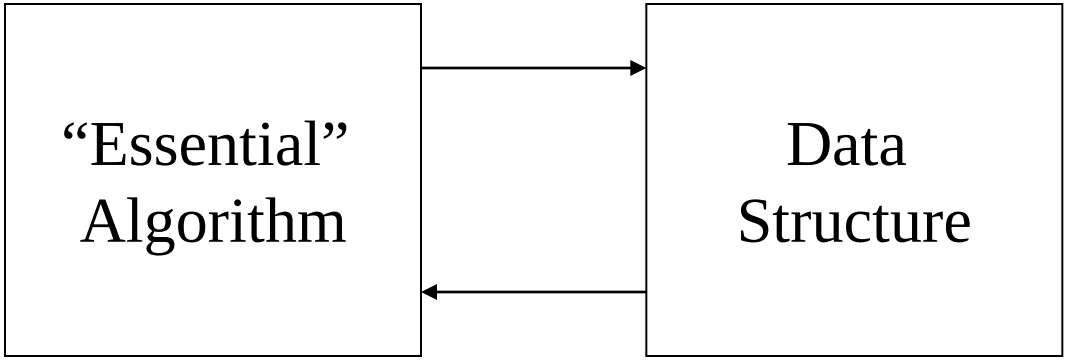




- Difficult, at least, to reuse or maintain data structure (for example).

Modular Approach

- Essential Algorithms for the application are carefully and clearly separated from the algorithms for manipulating the data structures used by the application.



- Easy to reuse or maintain just the data structure (for example)

Modular Approach (cont'd)

- The modular approach assists with:
  - managing complexity of solution
  - maintainability
  - correctness
  - reusability
  - distribution of workload (team of programmers)

Data Abstraction

- An abstract data type (ADT) consists of:
  1. An interface – a set of operations that can be performed
  2. The allowable behaviours – the way we expect instances of the ADT to respond to operations.

- The implementation of an ADT consists of:
  1. An internal representation – data stored inside the object's instance variables/members.
  2. A set of methods implementing the interface.
  3. A set of representation invariants, true initially and preserved by all methods

Interfaces

- A module interface specifies what the module user can do with the module.
- It defines the types, constants, and functions that are available, together with any restrictions and/or assumptions.
- The module user must use the module only via the provided interface, doing otherwise compromises the ability of the module author to change the module implementation.

IntList (simple array)

```
/* file: intlist-main.c
 */
#include <stdio.h>
#include <stdlib.h>

#ifdef INTLIST
#include INTLIST
#else
#error INTLIST must be defined with the quoted string header file name
#endif

#define SIZE 10

int main(void)
{
   IntList il;
   int i;
```

intList-main.c (cont'd)

```
if (MakeList(&il, SIZE) == FAILURE)
{
   fprintf(stderr, "MakeList(): failed\n");
   return EXIT_FAILURE;
}

/* fill the IntList with random numbers */
for(i=0; i<SIZE; i++)
{
   if (AddList(&il, rand()) == FAILURE)
   {
      fprintf(stderr, "AddList(): failed\n");
      break;
   }
}
```

intList-main.c (cont'd)

```
    printf("IntList size is %u\n", SizeList(&il));

    DisplayList(&il);

    FreeList(&il);

    return EXIT_SUCCESS;

}
```

intList-array.h

```
/* file: intlist-array.h
 *
 * IntList -- simple unordered array implementation
 *
 * type:  IntList
 *
 * constants: SUCCESS, FAILURE
 *
 * interface routines:
 *
 * int MakeList(IntList* pil, int size)
 *      attempts to initialise an IntList variable (passed by
 *      address) if insufficient memory is available for the size
 *      list requested then MakeList() retuns FAILURE, otherwise it
 *      returns SUCCESS
 *      MakeList() must be applied to an IntList before any other
 *      function.
```

intList-array.h (cont'd)

```
* void FreeList(IntList* pil)
*      attempts to reset an IntList variable (passed by address)
*      to the "empty" state, depending on the implementation this
*      may involve deallocation of memory. IntList must be
*      initialised with MakeList() again before use. Typically
*      FreeList() is the last function to be applied to an IntList
*
* int AddList(IntList* pil,int data);
*      attempts to add a new int (data) to an IntList variable
*      (passed by address). If the addition was successful
*      AddList() will return SUCCESS, otherwise FAILURE
*
* void DisplayList(IntList* pil);
*      displays all integers currently stored in the IntList
*      values displayed one per line on standard output
```

intList-array.h (cont'd)

```
*
* unsigned SizeList(IntList* pil);
*      returns the current size of the IntList
*      ie. how many data items are currently stored within
*      the list
*/
#define INTLISTSIZE 100
#define SUCCESS 1
#define FAILURE 0

typedef struct
{
  int array[INTLISTSIZE];
  unsigned size;
} IntList;
```

intList-array.h (cont'd)

```
int MakeList(IntList*,int);

void FreeList(IntList*);

int AddList(IntList*,int);

void DisplayList(IntList*);

unsigned SizeList(IntList*);

/*
 *
 */
```

intList-array.c

```
/* file: intlist-array.c
 *
 * IntList -- simple unordered array implementation
 */
#include <stdio.h>
#include "intlist-array.h"

int MakeList(IntList* pil,int size)
{
   if (size >  INTLISTSIZE)
   {
      return FAILURE;
   }
   pil->size = 0;
   return SUCCESS;
}
```

intList-array.c (cont'd)

```
void FreeList(IntList* pil)
{
   pil->size = 0;
}

int AddList(IntList* pil,int num)
{
   if (pil->size >= INTLISTSIZE)
   {
      return FAILURE;
   }
   pil->array[pil->size] = num;
   pil->size += 1;
   return SUCCESS;
}
```

intList-array.c (cont'd)

```
void DisplayList(IntList* pil)
{
   int i, size, *array;

   size = pil->size;
   array = pil->array;

   for(i=0; i<size; i++)
   {
      printf("%d\n", array[i]);
   }
}

unsigned SizeList(IntList* pil)
{
   return pil->size;
}
```

Portability

- Portability is widely considered a desirable attribute for the majority of programs regardless of size.
- Porting involves taking an existing program and modifying it so that it will execute on a new system/platform.
- Programs, through their design and implementation, can be more or less portable.
- The portability of a program is often linked to the extent to which a program uses platform specific functionality and/or the extent to which layers of abstraction around platform specific features were employed during the design and coding stages.

Portability (cont'd)

- Porting might involve "internationlisation" and/or localisation, and it might involve changes in implementation because of different underlying operating system, processor architecture, memory structure, and/or I/O devices on the target platform.
- Careful use of abstraction layers (incl. modular abstraction) can greatly improve the portability of a program.
- Many guides to portability in C/C++ will encourage you to use only a subset of the language, and most certainly not any non-standard features that some compiler vendors feel compelled to include with their product. For the sake of portability, avoid platform/compiler specific functions where ever possible.