Project *Rummikub*

Version 2

Deadline: May 6

(Wikimedia Commons)

In the second (and last) version of the project, the program will be modularized, and dynamic memory will be used in some of the data structures.

1. Modularization of the program

First, the program will be modularized according to the main data structures used. Besides the main program, the following modules will be created:

- Colors: it will include the type `tColor` and the associated subprograms.
- Tiles: it will include the type `tTile` and the associated subprograms.
- Bag: it will include the type `tBag` and the associated subprograms.
- Rack: it will include the type `tRack` and the associated subprograms.
- Racks: it will include the type `tRacks` and the associated subprograms.
- Sets: it will include the type `tSet` and the associated subprograms.
- Board: it will include the type `tBoard` and the associated subprograms.

2. Dynamic memory use

Dynamic memory will be used in the following elements of the game: bag, racks and sets.

The bag

In this version, the bag will be a bi-dimensional array of pointers to tiles created in dynamic memory. In the array positions of the tiles already dealt there will be a NULL, instead of a pointer to a tile with -1 as a number.

The racks

Each rack will be a list of tiles implemented with a dynamic array. The list will initially have a capacity for 8 tiles. When a new tile must be added to the list and the array is full, the capacity of the list will be incremented for 8 more tiles and when there are 8 free positions in the array, the capacity of the list will be reduced by 4 tiles. Remember that in order to modify the capacity of a list a new dynamic array must be created with the new capacity, the elements must be copied in the new array, and the old one must be deleted.

The sets and the board

In this version the sets will be implemented as linked lists (see the supplement of Lesson 4). And the board will be a static array of linked lists. Linked lists will be explained in detail.

3. Implementation details

It will be very important for the program to free all dynamic memory used.

- Module for the bag of tiles:
  The subprogram `createBag()` will create all tiles in dynamic memory.
  A new subprogram `deleteBag()` will free the dynamic memory used by the tiles remaining in the bag (it will be used at the end of the program).
- Module for individual racks:
  In the structure type `tRack` a new field `capacity` is needed, to know the current capacity of the tile list at every moment. A new procedure `initRack()` will initialize a rack, creating a dynamic array for 8 tiles. The tiles will no longer be directly added to the array of a rack, but a new procedure `newTile()` will add a tile to a rack by previously increasing the capacity of the list if the dynamic array is full.
  A subprogram `deleteRack()` will free the dynamic memory used by a rack.
  Increasing and reducing the capacity of a list based on dynamic arrays will be explained.
- In the module for the array of racks a new subprogram `deleteRacks()` will free the dynamic memory used by all players' racks. It will be used at the end of the program.
- As stated, the sets will be implemented as linked lists. A subprogram `copy()` will be needed to copy an existing set into a new set and another subprogram `deleteSet()` will free the dynamic memory used by a set.
- In the board module a new subprogram `deleteBoard()` will free the dynamic memory used by the sets added to the board. It will be used at the end of the program.

Other subprograms will need to be adapted to the new implementation of data structures.

A new option for the program's menu

Option 0 will now be option 5 (next player), and the new option 0 will allow the user to stop the game and exit the program. The names also have to be changed a little:

```
1: SortByNumber, 2: SortByColor, 3: SetsInRack, 4: PlaceTiles, 5: Next, 0: Exit >>>
```

Memory leaks detection

We need to add some code to be able to detect dynamic memory not freed at the end of the program (*memory leaks*). Add the following code at the beginning of the main program:

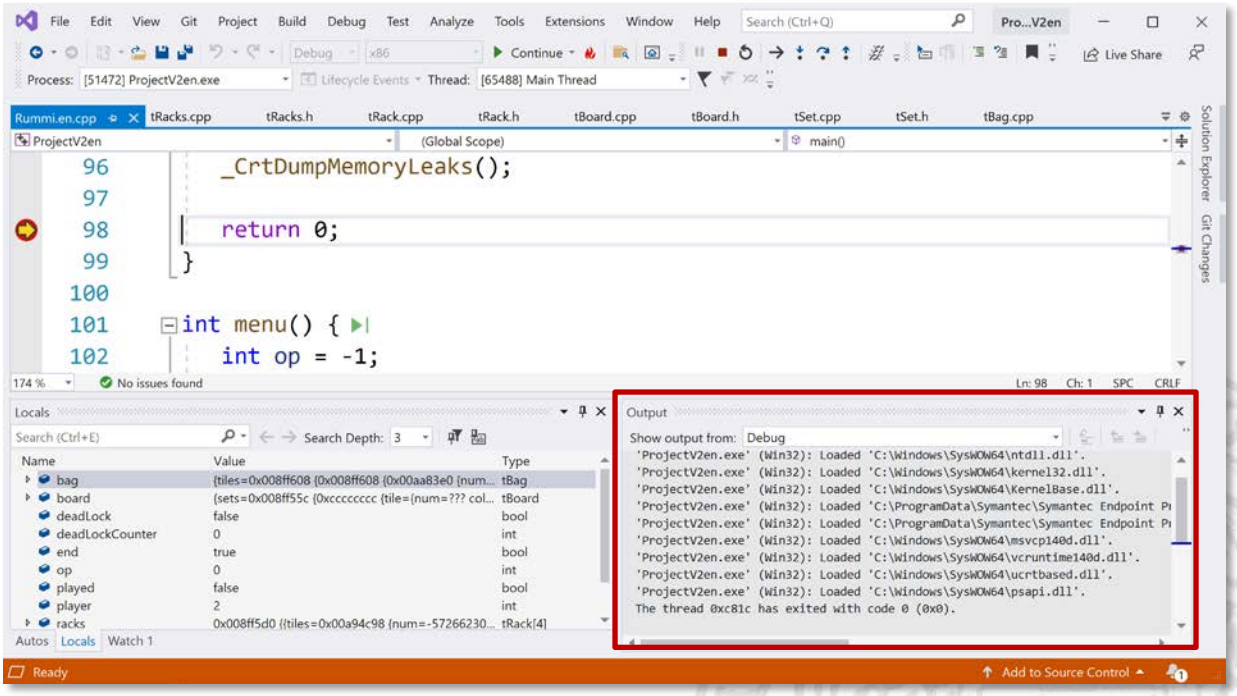
```
#define _CRTDBG_MAP_ALLOC
#include <crtdbg.h>
#include <cstdlib>
```

(Most likely the library `cstdlib` is already included in your program.) At the end of function `main()`, just before the instruction `return 0;` add the following call:

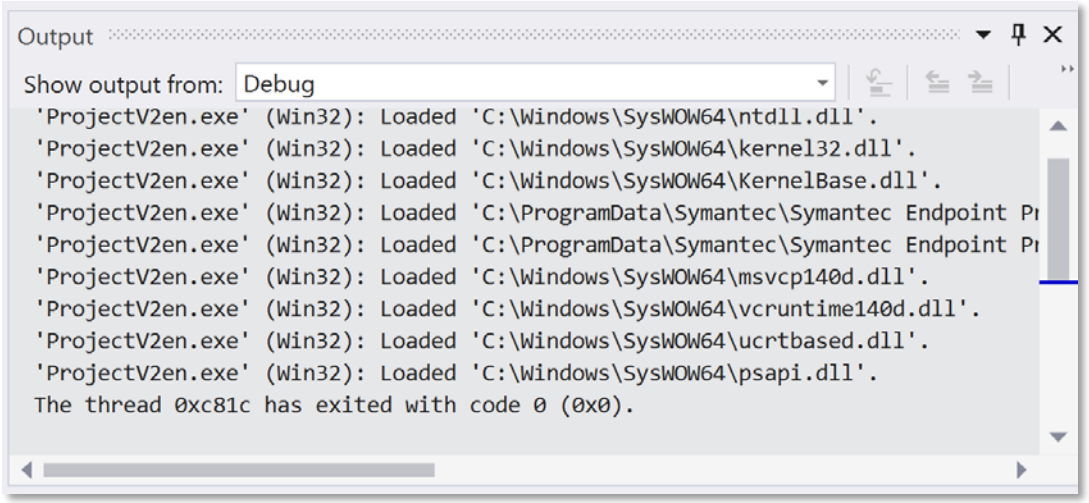
```
_CrtDumpMemoryLeaks();
```

Now you will be able to check if your program frees all dynamic memory used. How? By stopping program execution just before it ends and watching what is displayed in the Output panel in Visual Studio.

Set a breakpoint in the instruction `return` of `main()` and execute the program. Play a game (at least several turns) and when execution stops in the instruction `return` of `main()` check the panel Output, in which messages about program execution are displayed. Usually it is open below the editor (if it is not open, use the View menu to open it).



If there are no memory leaks, the panel Output will show a last message about a subprocess exited with code 0:



If some dynamic memory has not been correctly freed, then there will be messages informing of the situation: