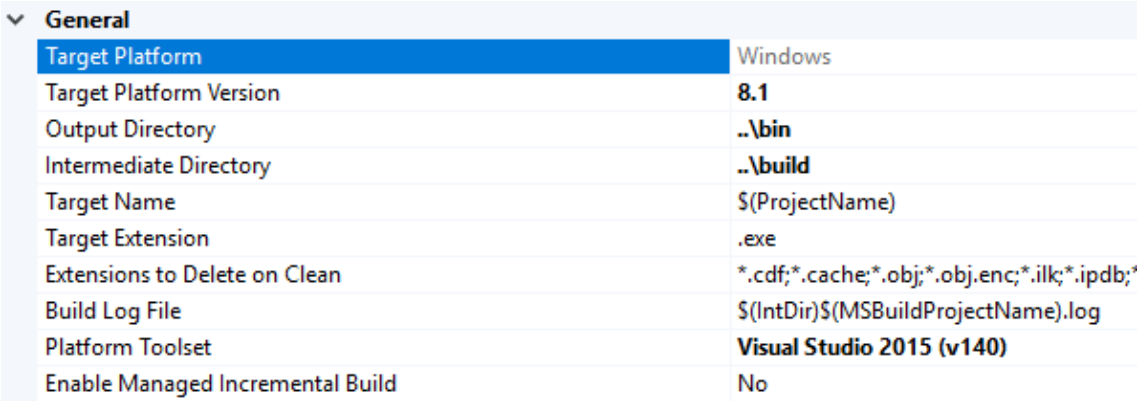
IA

ANÁLISIS TAD PORT

José María Martínez Carvajal

Visual Studio

Para una mayor comodidad a la hora de trabajar y facilitar la corrección de la migración se ha decidido trabajar sobre un proyecto de Visual Studio. Para poder mantener el directorio de trabajo establecido por Esat se han modificado las propiedades del proyecto de Visual Studio en todas sus configuraciones.



General	
Target Platform	Windows
Target Platform Version	**8.1**
Output Directory	**..\bin**
Intermediate Directory	**..\build**
Target Name	$(ProjectName)
Target Extension	.exe
Extensions to Delete on Clean	*.cdf;*.cache;*.obj;*.obj.enc;*.ilk;*.ipdb;*
Build Log File	$(IntDir)$(MSBuildProjectName).log
Platform Toolset	**Visual Studio 2015 (v140)**
Enable Managed Incremental Build	No

Migración TADS

La primera a tarea a realizar una vez el proyecto estaba listo y con su configuración adecuada fue la migración de las estructuras TAD a clases. Esto no fue un gran problema debido a que ya se había partido de un diseño de software pensado en clases en C.

Al realizar las funciones hubo tres casos específicos que dieron algo más de trabajo que el resto. Estos son la función Create, Destroy y Traverse.

En el caso de los Create debemos tener en cuenta que en C++ apareció la nueva palabra clave new vinculada con las clases. Por lo que ya no es recomendable utilizar la función malloc siempre y cuando no sea necesario y toda clase debería ser creada mediante la instrucción new. La función new tiene algunas peculiaridades no siempre conocidas para poder realizar todos los tipos de malloc que hemos realizado en C.

El malloc normal en el que reservamos una clase

```
Vector *vector = new Vector(5);
```

En caso de que queramos reservar memoria contigua

```
storage_ = new MemoryNode[capacity];
```

En caso de que queramos reservar void* (algo no muy común en C++)

```
data_ = operator new(bytes);
```

De la misma forma que en C++ es recomendable dejar de utilizar malloc a favor de new se recomienda utilizar la palabra reservada delete en lugar de free. Para el caso de free tenemos los mismos casos que new.

El free normal para liberar una clase

```
delete(vector);
```

En caso de que queramos liberar memoria contigua

```
delete[] storage_;
```

En caso de que queramos liberar void* (algo no muy común en C++)

```
operator delete(data_);
```

El problema con la función traverse es claro una vez nos paramos a pensar en como funciona C++ en las funciones de miembros. La mayor complejidad es que es complicado encontrar documentación sobre el tema, pues no es una práctica tan habitual. Sin embargo, el truco está en conocer que C++ tiene dos tipos de signatures para sus funciones, una para sus funciones estáticas y otra para las funciones no estáticas de clase.

Signature de una función en C

```
int (*pt2Function)(float, char, char) = NULL;
```

Signature de una función no estática de clase

```
int (TMyClass::*pt2Member)(float, char, char) = NULL;
```

Una vez sabemos esto debemos cambiar el parámetro callback de nuestra función traverse y saber como utilizarlo para invocar el callback:

```
traverse(&MemoryNode::print);

u16 Vector::traverse(s16(MemoryNode::*callback)()) const
{
	MemoryNode *aux = storage_;
	u16 i;
	for (i = 0; i< capacity_; i++) {
    (*aux.*callback)();
		++aux;
	}
	return i;
}
```

Posibles mejoras

Algunas posibles mejoras son las siguientes:

La lista podría ser más eficiente al extraer e insertar datos si se reservasen sus nodos con antelación, evitando de esta forma cambios de contexto y reduciendo los alojamientos de manera. Es una estrategia parecida a la seguida con el Memory Manager.

Cabe notar que la implementación de un vector circular ayudaría a bajar los tiempos del insert First, pues el header podría actualizarse hacía el final del vector tras varias inserciones seguidas.

También sería interesante ampliar el Memory Manager para que siempre tuviese constancia de la memoria que ha alojado, en estos momentos nuestro objetivo era acelerar la gestión de la memoria, pero sería interesante ampliarlo.