# 11-自己實作一個 vector 類別


- 11-1 規劃我們的 Vec 類別
- 11-2 實作 Vec 的細節
- 11-3 測試 Vec 類別
- 11-4 重載 [] 運算子
- 11-5 讓 Vec 可以儲存 int 以外的資料型別

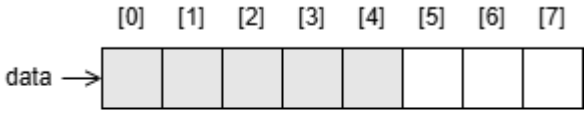
# 11-1 規劃我們的 Vec 類別

# 一、Vec 類別需要什麼？

在這個章節裡，我們嘗試自己建立一個簡單版的 vector，一個叫做 Vec 的類別。

我們至少需要以下三項屬性：

1. `data`: 指向一塊儲存資料的記憶體空間
2. `size`: 記錄目前 Vec 裡的元素數量
3. `capacity`: 記錄目前 Vec 裡已向作業系統要求配置的空間大小




size: 5

capacity: 8

接下來考慮需要對外公開的成員函數。

1. `Vec()`: 建構函數
2. `~Vec()`: 解構函數
3. `size()`: 回傳目前 Vec 裡有幾個元素
4. `capacity()`: 回傳目前 Vec 配置的記憶體大小最多可放幾個元素
5. `push_back(val)`: 將 val 附加在 Vec 的尾端
6. `pop_back()`: 將 Vec 尾端的元素移除

現在我們可以先完成這個類別的原型宣告如下:

**[vec.h]**

```
class Vec {
private:
    int *m_data;
    int m_size;
    int m_capacity;
public:
    Vec();
    ~Vec();
    int size();
    int capacity();
    void push_back(int val);
```

```
    void pop_back();
};
```

在成員函數的部分，我們使用 m_ 開頭的名稱，一方面用來提醒自己這是個 data member。一方面也是因為 data member 和 member function 不能用相同的名稱，我們不能有一個叫做 size 的 data member，同時又有一個叫做 size 的 member function。

# 11-2 實作 Vec 的細節

## 一、建構與解構函數

在建構函數中，我們要初始化 Vec 的 data member。因為剛建立好的 Vec 會是一個空的容器，所以一開始 m_size 和 m_capacity 都是 0。而 m_data 則先賦予它 nullptr 值，表示目前沒有指向任何記憶體。

```
Vec::Vec()
{
    m_data = nullptr;
    m_size = 0;
    m_capacity = 0;
}
```

在解構函數部分，我們先檢查是不有向作業系統要求配置記憶體，若有的話 m_data 將不會是 nullptr，而是會指向那塊記憶體。我們要釋放配置的記憶，把它還給作業系統。

```
Vec::~Vec()
{
    if(m_data!=nullptr)
    {
        delete [] m_data;
    }
}
```

## 二、size 和 capacity

Vec 的 size 和 capacity 會隨著 push_back 放入愈來愈多的元素而成長，這兩個 data member 不應該被使用者修改，所以我們只提供兩個相應的成員函數用來回傳其值。

```
int Vec::size()
{
    return m_size;
}
```

```
int Vec::capacity()
{
    return m_capacity;
}
```

# 三、push_back

在 Vec 空間需要成長時，我們採用一個簡單的策略 - 每次成長為之前的 2倍大。

要留意的是，由於一開始 capacity 是 0，乘以 2 之後還是 0，所以要特別處理這個狀況，在第一次 push_back 時，capacity 要成長為 1。

在成長的部分，流程如下。

1. 配置一塊原來 2 倍大的記憶體空間
2. 把舊資料搬到新空間
3. 釋放舊空間記憶體，還給作業系統
4. 將 m_data 指向新的這塊記憶體

```
void Vec::push_back(int val)
{
    // 如果空間不夠用
    if(m_size==m_capacity)
    {
        // 每次成長為原來的 2 倍大小
        int new_capacity = m_capacity*2;

        if(new_capacity==0)
            new_capacity = 1;

        // 配置一塊新的空間
        int *new_data = new int[new_capacity];
        // 把資料搬到新的空間
        for(int i=0; i<m_size; i++)
        {
            new_data[i] = m_data[i];
        }
        // 釋放舊的空間，改用新空間
        if(m_data!=nullptr)
            delete [] m_data;

        m_data = new_data;
        m_capacity = new_capacity;
    }

    m_data[m_size] = val;
    m_size++;
}
```

# 四、pop_back()

pop_back 部分很簡單，只要把 size - 1 即可，不必清除那塊記憶體。下次 push_back 時，新資料就會蓋掉它。

```
void Vec::pop_back()
{
    m_size--;
}
```

# 11-3 測試 Vec 類別

目前我們的 Vec 類別如下：

**[vec.h]**

```
#ifndef VEC_H_INCLUDED
#define VEC_H_INCLUDED

class Vec {
private:
    int *m_data;
    int m_size;
    int m_capacity;
public:
    Vec();
    ~Vec();
    int size();
    int capacity();
    void push_back(int val);
    void pop_back();
};

Vec::Vec()
{
    m_data = nullptr;
    m_size = 0;
    m_capacity = 0;
}

Vec::~Vec()
{
    if(m_data!=nullptr)
    {
        delete [] m_data;
    }
}

int Vec::size()
{
    return m_size;
}

int Vec::capacity()
{
    return m_capacity;
}
```

```
void Vec::push_back(int val)
{
    // 如果空間不夠用
    if(m_size==m_capacity)
    {
        // 每次成長為原來的 2 倍大小
        int new_capacity = m_capacity*2;

        if(new_capacity==0)
            new_capacity = 1;

        // 配置一塊新的空間
        int *new_data = new int[new_capacity];
        // 把資料搬到新的空間
        for(int i=0; i<m_size; i++)
        {
            new_data[i] = m_data[i];
        }
        // 釋放舊的空間，改用新空間
        if(m_data!=nullptr)
            delete [] m_data;

        m_data = new_data;
        m_capacity = new_capacity;
    }

    m_data[m_size] = val;
    m_size++;
}

void Vec::pop_back()
{
    m_size--;
}

#endif // VEC_H_INCLUDED
```

在主程式中引入其標頭檔來使用看看。

```
#include <iostream>
#include "vec.h"

using namespace std;

int main()
{
    Vec a;

    a.push_back(1);
```

```
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(3);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(5);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(7);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(9);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;

    return 0;
}
```

注意看 capacity 和 size 的變化。

Cap:1 Size:1 Cap:2 Size:2 Cap:4 Size:3 Cap:4 Size:4 Cap:8 Size:5

# 11-4 重載 [] 運算子

現在我們還缺一個重要的功能，那就是存取 Vec 裡的值。

試著執行這段程式看看。

**[main.cpp]**

```
#include <iostream>
#include "vec.h"

using namespace std;

int main()
{
    Vec a;

    a.push_back(1);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(3);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(5);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(7);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(9);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;

    for(int i=0; i<a.size(); i++)
    {
        cout << a[i] << " ";
    }
    cout << endl;

    return 0;
}
```

我們會得到這個錯誤訊息。

> error: no match for 'operator[]' (operand types are 'Vec' and 'int')|

因為在第 23 行中的`cout << a[i]`，編譯器不知道`Vec []`要怎麼處理。

`[]`是一個運算子，每種資料型別的`[]`運算可能有所不同，我們必須自己實作 Vec 的`[]`運算子運算。

在 Vec 宣告的尾端加入這行重載 `[]` 運算子的成員函數。

```
    ...
    void pop_back();
    int& operator[] (int index); // <== 加入這行
};
```

在函數實作部分加入。

```
int& Vec::operator[] (int index)
{
    return m_data[index];
}
```

這段程式碼是在做什麼呢? 我們把它拆解成四個部分來看。

- `int&`（回傳型態）：注意那個 & 符號，它代表「參照」（Reference）。意思是它不只傳回那個數字的值，而是直接把那塊記憶體的「真身」給你。這樣你才能寫出像 a[0] = 10; 這樣的程式碼，直接修改裡面的數值。
- `Vec::`（所屬類別）：這代表這個功能是屬於 Vec 這個類別。
- `operator[]`（函數名稱）：這是 C++ 的特殊關鍵字。組合起來就是告訴編譯器：「我要重新定義中括號 `[]` 的功能」。
- `(int index)`（參數）：中括號裡面填的數字（索引值）。例如 a[3]，這個 index 就是 3。

現在可以執行這段程式來驗證了。

**[main.cpp]**

```
#include <iostream>
#include "vec.h"

using namespace std;

int main()
{
    Vec a;

    a.push_back(1);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(3);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(5);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(7);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;
    a.push_back(9);
    cout << "Cap:" << a.capacity() << " Size:" << a.size() << endl;

    for(int i=0; i<a.size(); i++)
```

```
    {
        cout << a[i] << " ";  // 1 3 5 7 9
        a[i] = a[i]*2; // 把值修改為 2 倍
    }
    cout << endl;

    for(int i=0; i<a.size(); i++)
    {
        cout << a[i] << " ";  // 2 6 10 14 18
    }
    cout << endl;

    return 0;
}
```

Cap:1 Size:1 Cap:2 Size:2 Cap:4 Size:3 Cap:4 Size:4 Cap:8 Size:5 1 3 5 7 9 2 6 10 14 18

# 11-5 讓 Vec 可以儲存 int 以外的資料型別

目前我們 Vec 雖然可以動態成長，但是只能存放 int 型別的資料，這讓它變得很沒用。

明明只有型別不同，難道我們要寫一個 Vec_int 給 int 用，寫一個 Vec_double 給 double 用，……。

在這裡我們要使用一個新東西 `樣版(template)`，class 是用來產生物件的 [模版]，而 template 則是用來產生 class 的 [模版]。

你可以這樣想，在宣告時寫 `Vec<int>` 這時 template 就會幫我們生成一個可以儲存 int 的 class，宣告時寫 `Vec<double>` 這時 template 就會幫我們生成一個可以儲存 double 的 class。

在程式碼中插入 `template<typename T>` 表示接下來這個 類別、函數 裡，看到 `T` 時，把它替代成指定的型別。

例如：

```
template<typename T>
class Vec {
private:
  T* m_data;
  int m_size;
  int m_capacity;
};
```

想像在主程式中，我們宣告一個 `Vec<double> a;`，會自動產生一個像這樣的 Vec class。

```
class Vec<double> {
private:
  double* m_data;
  int m_size;
  int m_capacity;
};
```

如此一來，我們就可以讓 Vec 裡儲存各種型別的元素，甚至是我們自己設計的型別。

**修改後的 [vec.h]**

```
#ifndef VEC_H_INCLUDED
#define VEC_H_INCLUDED

template<typename T>
class Vec {
```

```
private:
    T *m_data;
    int m_size;
    int m_capacity;
public:
    Vec();
    ~Vec();
    int size();
    int capacity();
    void push_back(T val);
    void pop_back();
    T& operator[] (int index);
};

template<typename T>
Vec<T>::Vec()
{
    m_data = nullptr;
    m_size = 0;
    m_capacity = 0;
}

template<typename T>
Vec<T>::~Vec()
{
    if(m_data!=nullptr)
    {
        delete [] m_data;
    }
}

template<typename T>
int Vec<T>::size()
{
    return m_size;
}

template<typename T>
int Vec<T>::capacity()
{
    return m_capacity;
}

template<typename T>
void Vec<T>::push_back(T val)
{
    // 如果空間不夠用
    if(m_size==m_capacity)
    {
        // 每次成長為原來的 2 倍大小
```

```
        int new_capacity = m_capacity*2;

        if(new_capacity==0)
            new_capacity = 1;

        // 配置一塊新的空間
        T *new_data = new T[new_capacity];
        // 把資料搬到新的空間
        for(int i=0; i<m_size; i++)
        {
            new_data[i] = m_data[i];
        }
        // 釋放舊的空間，改用新空間
        if(m_data!=nullptr)
            delete [] m_data;

        m_data = new_data;
        m_capacity = new_capacity;
    }

    m_data[m_size] = val;
    m_size++;
}

template<typename T>
void Vec<T>::pop_back()
{
    m_size--;
}

template<typename T>
T& Vec<T>::operator[] (int index)
{
    return m_data[index];
}

#endif // VEC_H_INCLUDED
```