

14. 淺談記憶體管理

本章將就C++的記憶體管理，進行一個簡要的介紹：

在程式中的資料C++提供四種不同的記憶體管理方法：

- 自動儲存(automatic storage)
- 靜態儲存(static storage)
- 動態儲存(dynamic storage)
- 緒儲存(thread storage)

14.1 自動儲存

通常，在函式內所宣告的變數就是用“自動儲存”的方式，其宣告後會由編譯器負責為它配置一塊適合的記憶體空間，其空間的大小由其宣告的資料型態決定。在下面的程式碼中，變數a, b與temp皆屬之，所配置給它們的記憶體空間只有在foo函式的{ ... }範圍內存在，一但函式執行結束並返回時，這些空間就會被回收。換言之，變數能夠作用的範圍就侷限在其宣告所在的函式裡。

```
int foo(int a, int b)
{
    int temp;
    temp = 2*a+3*b;
    return temp;
}
```

由於此類變數在程式執行時，其在記憶體裡的空間配置與回收都是“自動”完成的，所以又被稱為「自動變數(Automatic Variable)」更具體來說，當一個程式經編譯後執行時，會得到一塊由作業系統所配置的空間(以Linux系統為例，每個執行中的程式可被配置到4GB的虛擬空間)，而在程式中的自動變數將會使用程式所配置到的記憶體空間裡的Stack區段，依後進先出(Last-In First OutLIFO)的順序加以管理——依變數宣告的順序配置Stack區段的空間，並以相反的順序從Stack中移除。自動變數的生命週期限於其所位於的程式區塊中，例如上述程式foo函式由{開始至}結束的範圍就稱為一個**程式區塊(block)**。我們也可以在程式中視需要建立巢狀的程式區塊，例如：

```
int main()
{
    int i, j;
    scanf(" %d", &i);
    scanf(" %d", &j);
    {
        int x;
```

```
        x=2*i;
        if(j>x)
            j=x;
    }
}
```

在這個例子中，`main()`函式因為某些原因，暫時地需要一個變數`x`以進行相關的計算，因此，我們將需要`x`的地方標記為一個程式區塊，這樣一來，在這個區塊之外，就不存在`x`這個變數。若沒有了區塊，變數`x`將持續存在於記憶體內，直到`main()`函式結束為止。

14.2 靜態儲存

用`static`的方式修飾變數的宣告，還是由編譯器配置其所需的記憶體空間，但是是放在不同於自動變數所在的`Stack`區段，且其生命週期橫跨整個程式的範圍，例如：

```
static int counter = 0;
```

14.3 動態儲存

動態儲存是以`new`來配置記憶體空間，並以`delete`來將空間刪除(事實上，記憶體空間不會被刪除的，只會被釋放並回收)，往往是搭配指標使用。

14.3.1 指標與動態記憶體管理

通常，我們都是用類似下面的程式碼的方式，在使用指標：

```
int *p;
int x;

p=&x;
```

在`C++`中，提供一個新的方式：

```
int *p;

p = new int;

或是

int *p = new int;
```

用這種方式，直接取得在記憶體中的一塊位置，而不需要另外宣告一個整數。同樣的，當你不再需要這塊位置時，可以`delete`將其刪除(意即，還給系統)。

```
int *p = new int;

delete p;
```

14.3.2 動態陣列

我們可以透過`new`與`delete`來動態地建立與刪除(或者說回收)陣列。請參考下面的例子：

```
int *p = new int [10]; // 動態配置產生一塊存放10個整數的陣列空間，並讓p指向它。

p[0]=3;
p[1]=2;

delete [] p;           // 不再使用時將其刪除
```

14.3.3 動態結構體

我們也可以透過`new`與`delete`來動態地建立與刪除(或者說回收)結構體。請參考下面的例子：

```
typedef struct
{
    int x, y;
} Point;

Point *p = new Point;
```

但是要注意的是，透過指向結構體的指標來存取其內部的資料欄位時，必須要使用`->`，而不是使用`.`；或是使用間接存取的方式，先取得指標所指向的結構體，然後就可以使用`.`來存取，例如：

```
p->x=5;    // 透過指標，使用 -> 存取欄位
(*p).y=6; // 先間接取得"值"以後，再存取欄位
```

當不再使用該結構體時，也可以使用`delete`來刪除指標所指向的結構體：

```
delete p;
```

或是

```
struct point
{
    int x, y;
};

point *p = new point;
p->x=5;
(*p).y=6;

delete p;
```

14.3.4 動態結構體陣列

下面的程式範例，宣告並動態配置了結構體的陣列：

```
#include <iostream>
using namespace std;

struct point
{
    int x, y;
};

int main()
{
    point *pdata = new point [10];

    for(int i=0;i<10;i++)
    {
        pdata[i].x=pdata[i].y=i;
    }

    cout << "pdata[0].x = " << pdata[0].x << endl;
    cout << "pdata[0].x = (*pdata).x = " << (*pdata).x << endl;
    cout << "pdata[2].x = " << pdata[2].x << endl;

    //  pdata++;
    cout << "(pdata+2)->x = " << (pdata+2)->x << endl;
    cout << "(*(point *) (pdata+3)).x = " << (*(point *) (pdata+3)).x << endl;
    cout << "(*pdata).x = " << (*pdata).x << endl;
```

```
delete [] pdata;
return 0;
}
```

再看看下面這個二維結構體陣列的例子：

```
#include <iostream>
using namespace std;

#define ROW 3
#define COL 2

struct point
{
    int x, y;
};

int main()
{
    // 宣告並動態配置一個二維的point陣列
    // declare a two dimensional array of points
    // , i.e., point[ROW][COL].
    // by using an array of pointers to arrays.
    // It has been newed in a loop.

    point **p2d = new point *[ROW];

    for(int i=0;i<ROW;i++)
    {
        p2d[i]=new point [COL];
        for(int j=0;j<COL;j++)
        {
            p2d[i][j].x = p2d[i][j].y = (i+1)*(j+1);
        }
    }

    cout << "p2d[i][j]=" << endl;
    cout << "i\\j\\t0  |\\t1  |" << endl;
    cout << "---+-----+-----+" << endl;
    for(int i=0;i<ROW;i++)
    {
        cout << " " << i << " | ";
        for(int j=0;j<COL;j++)
            cout << "(" << p2d[i][j].x << "," << p2d[i][j].y << ") | ";
        cout << endl;
    }
}
```

```

cout << endl;
cout << "p2d[0][0].x = (*p2d)[0].x = " << (*p2d)[0].x << endl;
cout << "p2d[0][1].x = (*p2d)[1].x = " << (*p2d)[1].x << endl;
cout << "p2d[2][0].x = (*(p2d+2))[0].x = " << (*(p2d+2))[0].x << endl;

cout << endl;
cout << "p2d[2][0].x = (*(p2d+2))->x = " << (*(p2d+2))->x << endl;
cout << "p2d[2][1].x = (*(p2d+2)+1)->x = " << (*(p2d+2)+1)->x << endl;

for(int i=0;i<ROW;i++)
    delete [] p2d[i];
delete [] p2d;

return 0;
}

```

其執行結果如下：

```

[11:57 user@ws ch13]$ ./a.out
p2d[i][j]=
ij|    0    |    1    |
---+-----+-----+
0 | (1,1) | (2,2) |
1 | (2,2) | (4,4) |
2 | (3,3) | (6,6) |

p2d[0][0].x = (*p2d)[0].x = 1
p2d[0][1].x = (*p2d)[1].x = 2
p2d[2][0].x = (*(p2d+2))[0].x = 3

p2d[2][0].x = (*(p2d+2))->x = 3
p2d[2][1].x = (*(p2d+2)+1)->x = 6
[11:57 user@ws ch13]$

```

14.3.5 動態字串

你也可以使用new及delete來動態地建立與回收字串所需的記憶體空間，請參考下面這個程式：

```

#include <iostream>
using namespace std;
#include <cstring>

char * getUsername(void);

int main()
{
    char *name;

```

```
name=getUserName();
cout << "Hello, " << name << "!\n";
delete [] name;

return 0;
}

char * getUserName()
{
    char temp[80];
    cout << "Enter your name: ";
    cin.getline(temp, 80);
    // cin >> temp;

    char *pName = new char[ strlen(temp) + 1];
    strcpy(pName, temp);

    return pName;
}
```

14.4 緒儲存

緒儲存主要是應用在多執行緒(multithreading)程式設計中，我們在此暫不加以說明。

From:

<https://junwu.nptu.edu.tw/dokuwiki/> - Jun Wu的教學網頁

國立屏東大學資訊工程學系

CSIE, NPTU

Total: 119197

Permanent link:

<https://junwu.nptu.edu.tw/dokuwiki/doku.php?id=cppbook:ch-memallocation>



Last update: **2024/02/19 12:48**