What's New?

參考資料:

A Tour of C++, by Bjarn Stroustrup

http://www.stroustrup.com/Tour.html

主題: C++ 簡介

這個單元的主要任務介紹 C++，讓大家能夠開始練習用 C++ 寫程式。C++涵蓋的內容非常廣，我們這學期剩下的時間只能認識到其中的一小部分。但是，我們也沒必要把全部的東西都學過才能寫程式，學習程式設計是一個漸進的過程(http://norvig.com/21-days.html) ，應該一邊學一邊寫，而且要培養自學能力。有些瑣碎的東西如果都在課堂上講解，其實聽起來也會很枯燥。我們還是會照著這學期的課程規劃，試著把資訊系統導論和程式設計兩門課結合起來，透過寫程式來認識資工的基本知識，而學習C++只是其中的一部分。

因為我們已經學過一個學期的 C，所以我們就先以 "What's New?" 做為主軸，來認識 C++。這個標題除了有 "比較 C ++ 和 C 的不同之處" 的意涵之外，也想點出 C++ 最近幾年有甚麼進展和演變。我們還是都採取舉例說明的方式來介紹，介紹的內容會比較片面，不足的地方要自行靠 Google 補足。(譬如我們不需要像剛學 C 的時候，連 printf 的各種格式如何使用都要介紹。)

第一件事還是 Hello, World! ，在 C++ 是這樣寫：

```
#include <iostream>
int main()
{
    std::cout << "Hello, World!\n";
}
```

可以看到不同的地方是 `#include<stdio.h>` 變成了 `#include <iostream>`，然後是用 `std::cout` 來輸出。 `std::` 是用來指定 cout 的 namespace 屬於 standard library，這樣的用意是為了避免同名的衝突。如果在程式碼前面寫 `using namespace std;` 則之後使用 standard library namespace 底下的東西時，就可以省略 開頭的 `std::`。

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, World!\n";
}
```

string

在 C++ 裡面，有 string 資料型態可用 (屬於 standard library 所以可以視為內建)，複製 string 比以前在寫 C 程式時簡單很多，用 + 就行了。

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string name, ans;
    cout << "Hi! My name is Hal." << "What's your name?\n";
    cin >> name;
    ans = "Nice to meet you, " + name + ".";
    cout << ans << endl;
    cout << "Your name has " << name.size() << " characters." << endl;
}
```

auto

C++ 可以將變數的型別設為 auto，編譯器會替你做 auto type inference，依照前後文決定變數的型別，例如 `auto i = 5;` 或是 `auto x = 2.3;` 可以知道 `i` 的型別應該就是 `int` 而 x 的型別會是 `double`。

Range-based for loop

請看底下的範例

```
#include <iostream>
#include <vector>
#include <iomanip>
using namespace std;
int main()
{
    vector<int> v{1,2,3,4,5};
    for (auto i : v)
        cout << setw(3) << i;
}
```

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<string> vs{"Tom", "John", "Amy", "Cathy"};

    for (auto n : vs)
        cout << n << endl;
}
```

`vector` 是 standard library 裡面定的 container，`vector<int> v;` 表示 v 是由 `int` 構成的 vector，每個元素的型別都是 `int`。同理 `vector<string> vs;` 則每個元素都是 `string`。在 C++ 可以用以前的 C for loop 語法，但除此之外，C++ 也提供 range-based for loop，請看範例中 for 的用法，會循序取出 vector 裡面的每個元素。這樣的寫法搭配 `auto` 變數，可以當成片語來使用，比起以往在 C 語言中必須用額外的 index 存取 array 元素的寫法，來得簡潔許多，也不必擔心 index 超出陣列範圍。

Reference &

C++ 多了一種 變數類型，叫做 reference。通常是用在函數傳遞參數，可以達到 call-by-reference 的效果。回憶一下，在 C 語言裡面的函數呼叫，都是 call-by-value，函數呼叫時傳遞的其實是參數的值 (將值放進 Stack)，所以如果在函數中對參數做修改，不會影響到原本的變數 (底下 swap 的例子)。也因為如此，C 必須利用指標變數，傳遞記憶體位址，來達到 call-by-reference 的效果。在 C++ 則是直接提供 能夠 達到 call-by-reference 的機制。對照底下兩個範例的差異。

```
#include <iostream>
using namespace std;
void swap(int i, int j)
{
    int t = i;
    i = j;
    j = t;
}
int main()
{
    auto x = 5, y = 7;
    cout << "(" << x << "," << y << ")\n";
    swap(x, y);
    cout << "(" << x << "," << y << ")\n";
}
```

```
#include <iostream>
using namespace std;
void swap(int& i, int& j)
{
    int t = i;
    i = j;
    j = t;
}
int main()
{
    auto x = 5, y = 7;
    cout << "(" << x << "," << y << ")\n";
    swap(x, y);
    cout << "(" << x << "," << y << ")\n";
}
```

差別只是參數的型別從 int 變成 int &。Reference 相當於是替變數取了一個代稱，所以在 swap 函數裡面的 i 其實就相當於是 main 裡面的 x，而 j 則相當於 y。

除了參數，回傳值也可以用 Reference。底下的程式碼，右邊的寫法是錯的。原因是當函數結束之後，函數內原本記錄在 Stack 中的局部變數都不能再被使用，所以把局部變數的 reference 傳回去是沒有意義的，會造成程式錯誤。此外，如果不希望傳回來的 Reference 被修改，可以把函數改成 const int& max(int& i, int& j)，這樣一來如果使用 max(x,y)=0;，在編譯時編譯器就會提出錯誤訊息讓程式設計者知道用法不對。

```
#include <iostream>
using namespace std;
int& max(int& i, int& j)
{
    return (i>j) ? i : j;
}
int main()
{
    auto x = 5, y = 7;
    cout << x << " " << y << "\n";
    max(x, y) = 0;
    cout << x << " " << y << "\n";
}
```

```
#include <iostream>
using namespace std;
int& max(int& i, int& j)
{
    int m;
    m = (i>j) ? i : j;
    return m;
}
int main()
{
    auto x = 5, y = 7;
    cout << x << " " << y << "\n";
    max(x, y) = 0;
    cout << x << " " << y << "\n";
}
```

Lambda functions

C++11 將 lambda functions 納入標準中，所以現在 C++ 也可以寫出沒有名字的函數，讓某些應用變得方便很多。先看下面的例子：

```
// -std=c++11
#include <algorithm>
#include <iostream>
#include <vector>
#include <iomanip>
using namespace std;
int main()
{
    vector<int> v;
    for (int i = 1; i < 10; ++i) v.push_back(i*5);
    for (auto i : v) cout << setw(5) << i;
    cout << endl;

    for_each(v.begin(), v.end(), [] (int n) {
        if (n % 2 == 0) {
            cout << n <<" is even " << endl;
        }
    });

    for (const auto & y : v) cout << setw(5) << setfill('#') << y;
    cout << endl;
    for (auto & y : v) y = y+1;
    for (auto y : v) cout << setw(5) << setfill('*') << y;
    cout << endl;
}
```

首先， #include 多了好幾個 header 需要引入，其中 <algorithm> 是為了後面要用到的 for_each，<iomanip> 則是為了設定 cout 格式。 程式碼一開始的 for 利用 vector 的 push_back 函數把資料放入 v 裡面，之後再把 v 的內容一一顯示出來，其中 setw(5)是 <iomanip> 提供的函數，作用是將 cout 的輸出格式設定為寬度等於 5，所以顯示出來的每筆數字都會占5個字元，不足的會補上空白。接下來的 for_each 函數，用來將 vector 的每個元素，一一代入指定的函數中。for_each的前兩個參數用來指定 vector需要被處理的範圍，分別代表開始和結束的位置。for_each 的第三個參數則是要被代入的函數，在範例中就是用 lambda function 的方式，直接定義要被代入的函數，語法如下，看起來和一般的函數沒甚麼不同，只是不需要取名字，只要寫 [] 就行了

```
[] (int n) {
    if (n % 2 == 0) {
        cout << n <<" is even " << endl;
    }
}
```

More lambda functions (先自行研究，過幾周之後我們會再來解讀)

```
#include <iostream>
#include <functional>
#include <math.h>
#include <iomanip>
#define TOL 0.00001
using namespace std;
void print(function<double(double)> f, double x)
{
    cout << "f(" << fixed << setprecision( 5 )  << x << ") = " << f(x) << endl;
}
void print(double x)
{
    cout << "ans = " << x << endl;
}
double fixed_point(function<double(double)> f, double guess)
{
    auto close_enough = [](double v1, double v2)
        { return fabs(v1-v2)<TOL; } ;
    auto next = f(guess);
    if (close_enough(next, guess)) return next;
    else return fixed_point(f, next);
}
double mysqrt(double x)
{
    return fixed_point([x](double y){return (y+(x / y))/2.0; }, 1.0);
}
double mycbrt(double x)
{
    return fixed_point([x](double y){return (y+(x / (y*y)))/2.0; }, 1.0);
}
int main()
{
    auto x = 2.0;
    print(mysqrt, x);
    print(mycbrt(5.0));
}
```

Classes

C++和C最大的差別，應該就是多了 class，讓使用者能夠自訂型別並且運用物件導向的概念來設計程式。 我們底下介紹的是最經典的範例之一，程式碼取自 A Tour of C++ 這本書。C++ Standard Library 裡面已經有提供 `complex` 的型別，我們用的範例是它的簡化版。在其他C++的書裡面也可以找到類似的版本。

`class` 的語法有點類似 `C` 語言裡面的 `struct`，但是在 `C++` 裡面我們自己定義的 `class` 直接就能拿來當作新的型別來使用，例如

```
class complex
{
 ...
} ;
```

這樣就定出了一個新的型別叫做 complex，要注意最後面波浪括號之後還有一個分號。我們用這個新的型別來宣告變數，例如

```
complex z;
```

這樣 `z` 就是一個 `complex` 變數，通常我們稱它為 物件。

C++ Class 的特色是把資料和函數綁在一起，class 裡面所定義的成員可以包含資料和函數，讓物件導向程式設計的概念得以實現。Class 不只是把資料包裝起來，通常也會把處理資料所需的函數包含進來，譬如前幾頁的例子裏面的 `name.size()` 或是 `v.push_back(i*5)`，其中 name 是物件，`size()` 是它的成員函數，v 是物件，push_back()是成員函數。C++ class 的成員 預設都是 private，意思是無法從 class 外部存取，只有那個 class 自己的成員可以使用 private 區域的資料或是函數。如果要開放給外部使用，則要在 class 裡面另外再定一個區塊叫做 public:，寫在 public 區塊的成員資料或是成員函數，可以被 class 以外的函數或是其他物件存取。放在 public 區塊裡的函數，提供了 class 和外界溝通的管道。

下兩頁的範例涵蓋了 C++ class 的基本定義方式，裡面有許多細節我們會在接下來幾周逐一介紹。剛開始的時候，可以先試著依樣畫葫蘆，自己試著定義其他的 class，譬如模仿 complex ，定義出有理數 rational 型別，(complex 有實部和虛部，有理數則是有分子和分母)，這會是很好的練習，對於建立物件導向的概念以及熟悉語法會有些幫助。

```
#include <iostream>
//#include <vector>
#include <iomanip>
using namespace std;

class complex
{
    double re, im;
public:
    complex(double r, double i): re {r}, im {i} { }
    complex(double r): re {r}, im {0} { }
    complex(): re {0}, im {0} { }

    double real() const
    {
        return re;
    }
    void real(double d)
    {
        re=d;
    }
    double imag() const
    {
        return im;
    }
    void imag(double d)
    {
        im=d;
    }

    complex& operator+=(complex z)
    {
        re+=z.re, im+=z.im;
        return *this;
    }
    complex& operator-=(complex z)
    {
        re-=z.re, im-=z.im;
        return *this;
    }
    complex& operator*=(complex);
    complex& operator/=(complex);

    friend std::ostream & operator<<(std::ostream& os, const complex& a);

};
```

接上頁

```
complex& complex::operator*=(complex z)
{
    double tre = re;
    double tim = im;
    re = tre*z.re - tim*z.im;
    im = tre*z.im + tim*z.re;
    return *this;
}
complex& complex::operator/=(complex z)
{
    double tre = re;
    double tim = im;
    double norm = z.re*z.re + z.im*z.im;
    re = (tre*z.re + tim*z.im)/norm;
    im = (tim*z.re - tre*z.im)/norm;
    return *this;
}
complex operator+(complex a, complex b) { return a+=b; }
complex operator-(complex a, complex b) { return a-=b; }
complex operator-(complex a)
{
    return {-a.real(), -a.imag()};
}
complex operator*(complex a, complex b) { return a*=b; }
complex operator/(complex a, complex b) { return a/=b; }
bool operator==(complex a, complex b)
{
    return a.real()==b.real() && a.imag()==b.imag();
}
bool operator!=(complex a, complex b) { return !(a==b); }
ostream & operator<<(ostream& os, const complex& a)
{
    if (a.imag()>0)
        os << a.real() << "+" << a.imag() <<  "i";
    else if (a.imag()<0)
        os << a.real() <<  a.imag() << "i";
    else
        os << a.real();
    return os;
}
int main()
{
    complex a {2.3};
    complex b {2.5, -4.2};
    complex c {a+complex{1,2.3}};

    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    if (a!=b) a+= b;
    cout << a << endl;

}
```