แนะนำภาษา C++

เทพพิทักษ์ การุญบุญญานันท์

ภาษา C++

- C++ = ภาษา C ที่ปรับปรุง syntax และเพิ่มการรองรับ OOP
- เป้าหมายของ C++
  - compatible กับ C
  - native binary code ($\neq$ VM code)
    $\rightarrow$ close to machine
  - object-oriented programming
    $\rightarrow$ close to problem being solved
  - scalable (พัฒนาโปรแกรมได้ทุกขนาด)

ตัวอย่างโปรแกรมที่เขียนด้วย C++

- LibreOffice (ชุดสำนักงาน – เวิร์ด, สเปรดชีต, งานนำเสนอ)
- Inkscape (vector graphics คล้าย Adobe Illustrator)
- Qt (GUI toolkit สำหรับเดสก์ท็อปและ smart phone)
- VLC (media player, streamer)
- Squid (HTTP proxy)
- MySQL (RDBMS)
- Mozilla Firefox (เว็บเบราว์เซอร์)
- Google Chrome (เว็บเบราว์เซอร์)
- GRASS (ระบบสารสนเทศภูมิศาสตร์)
- Celestia (โปรแกรมจำลองดาราศาสตร์)

จาก C เป็น C++ : สิ่งเล็กๆ น้อยๆ

- line comment

```
/* Multi-line
 * C comment
 */

// C++ line comment
```

- ชนิด **bool** พร้อมค่า **true**, **false**

```
bool is_done = false;
while (!is_done) {
   ...
}
```

จาก C เป็น C++ : สิ่งเล็กๆ น้อยๆ

- ประกาศตัวแปรเมื่อไรก็ได้ (ไม่จำเป็นต้องเป็นที่ต้นบล็อค)

```
int i = 0;
print_val (i);
int j = i + 3;
print_val (j);
```

จาก C เป็น C++ : สิ่งเล็กๆ น้อยๆ

- ตัวแปรที่ต้นลูป **for**

```
for (int i = 0; i < 10; i++)
   ...
```

มีความหมายเทียบเท่า:

```
{
   int i = 0;
   while (i < 10) {
      ...
      i++;
   }
}
```

จาก C เป็น C++ : สิ่งเล็กๆ น้อยๆ

- ชื่อ **struct**, **union**, **enum**, **class** สามารถใช้เป็นชื่อชนิดได้ทันที

```
struct Student { ... };
union Value { ... };
enum DOW { ... };
...
Student rec;
Value   val;
DOW     dow;
```

- ลดความยาวของซอร์สโค้ดลง
- สะดวกในการกำหนดชนิดใหม่โดยไม่ต้องคอย **typedef**
- **class** จะกล่าวถึงต่อไป

จาก C เป็น C++ : สิ่งเล็ก ๆ น้อย ๆ

- inline function

```
inline int max (int a, int b) { return a > b ? a : b; }
```

  - ใน binary จะไม่มีการเรียกฟังก์ชัน แต่จะแทนโค้ดลงไปเลย
  - ใช้กับฟังก์ชันเล็ก ๆ เพื่อประหยัด cost ของ function call

- ค่าคงที่ที่สามารถใช้ขณะคอมไพล์ได้

```
const int MaxLen = 80;
char name[MaxLen];
```

inline function + compile-time const → ลดการใช้ **#define**

จาก C เป็น C++ : new และ delete

- การใช้หน่วยความจำแบบ dynamic ด้วย **new** และ **delete**

```
int   *p = new int;
char *s = new char[10];
Student *rec = new Student (10, "Julie", 60);
...
delete p;
delete[] s;
delete rec;
```

- เชื่อมรวมกับตัวภาษาได้ดีกว่า **malloc()** และ **free()**
- สามารถกำหนดค่าเริ่มต้นขณะ **new** ได้
- มีการเรียก constructor ขณะ **new** และเรียก destructor ขณะ **delete**
- หาก **new** เป็นแอร์เรย์ ควรทำลายด้วย **delete[]**

จาก C เป็น C++ : Reference

- เป็นการสร้าง alias ของตัวแปรหรือข้อมูล

```
int  x = 0;
int& a = x;   // a is now an alias of x
a = 1;        // x is now 1
```

- ความแตกต่างจากพอยน์เตอร์
  - reference ใช้อ้างถึงตัวแปรหรือข้อมูลอื่นได้ทันทีโดยไม่ต้อง dereference
  - reference หลังจากสร้างแล้วไม่สามารถเปลี่ยนไปอ้างถึงตัวแปรหรือข้อมูลอื่นได้อีก
  - reference ต้องมีการกำหนดค่าเริ่มต้นเสมอ

จาก C เป็น C++ : Reference

- ประโยชน์ของ reference
  - call by reference

```
void swap (int& a, int& b)
{
  int tmp = a;
  a = b;
  b = tmp;
}
```

```
int x = 1;
int y = 2;
swap (x, y);   // now x == 2 and y == 1
```

จาก C เป็น C++ : Reference

- ประโยชน์ของ reference
  - ใช้ส่งอาร์กิวเมนต์ที่เป็นออบเจกต์ใหญ่ ๆ ให้กับฟังก์ชันโดยไม่มีการ copy

```
void printObjSlow (Student s) { /* ... */ }
void printObjFast (const Student& s) { /* ... */ }
```

```
Student s1;
// slow, s1 is copied to argument s
printObjSlow (s1);
// fast, s gets direct read-only access to s1
printObjFast (s1);
```

จาก C เป็น C++ : Reference

- ประโยชน์ของ reference
  - ใช้คืนค่าออบเจกต์เพื่อให้ผู้เรียกสามารถนำไปใช้ต่อได้

```
Output& Output::PrintInt (int i)
  { printf ("%d", i); return *this; }
Output& Output::PrintText (const char* text)
  { printf ("%s", text); return *this; }
```

```
Output out1;
out1.PrintInt(2).PrintText(",").PrintInt(3);
```

(ผู้เรียนจะเข้าใจไวยากรณ์เมื่อได้เรียนเกี่ยวกับคลาส)

จาก C เป็น C++ : Function Overloading

- ฟังก์ชันในภาษา C++ สามารถมีชื่อซ้ำกันได้ถ้ารับอาร์กิวเมนต์ต่างกัน
- เรียกว่าการ overload ฟังก์ชัน

```
int    sum (int a, int b)          { return a + b; }
double sum (double a, double b)    { return a + b; }
```

```
int    a = sum (1, 2);
double d = sum (1.0, 2.0);
```

จาก C เป็น C++ : Function Default Arguments

- ฟังก์ชันในภาษา C++ สามารถละอาร์กิวเมนต์บางตัวขณะเรียกได้
- อาร์กิวเมนต์ที่จะละได้
  - ต้องมีการกำหนดค่าปริยายเมื่อละ
  - ห้ามมีอาร์กิวเมนต์ที่ไม่มีค่าปริยายต่อท้ายอีก

```
void print (char c, int copies = 1);
```

```
print ('*');     // equivalent to print ('*', 1);
print ('*', 5);
```

จาก C เป็น C++ : Function Default Arguments

- การใช้งาน
  - ใช้เพิ่มอาร์กิวเมนต์ของฟังก์ชันในภายหลังเพื่อเพิ่มรายละเอียด
    → ไม่กระทบโค้ดเดิมที่เรียกแบบเดิม
  - ใช้สร้างฟังก์ชันที่ทำงานแบบเจาะจงหลายระดับ
  - ถ้าเป็นการเพิ่มอาร์กิวเมนต์ที่ทำให้ความหมายเปลี่ยน
    → ควร overload ฟังก์ชันแทน

จาก C เป็น C++ : Namespace

- เป็นการแยกขอบเขตของชื่อต่าง ๆ ออกเป็นส่วน ๆ ไม่ให้ชนกัน

```
namespace foo {
  int getHeight() { return 5; }
}

namespace bar {
  double getHeight() { return 3.14; }
}
```

```
int    ih = foo::getHeight();
double dh = bar::getHeight();
```

จาก C เป็น C++ : Namespace

- สามารถใช้ไดเรกทีฟ **using** ในการประกาศใช้ namespace ในโค้ดที่ตามมา
  - **using** กับทั้ง namespace

```
using namespace foo;
int ih = getHeight();   // calls foo::getHeight()
```

  - **using** แบบเจาะจงชื่อ

```
using foo::getHeight;
int ih = getHeight();   // calls foo::getHeight()
```

จาก C เป็น C++ : Namespace

- namespace สามารถกำหนดทีละส่วนแยกกันได้

```
namespace A { // namespace for users
  void f();
}

#include <stdio.h>

namespace A { // namespace for implementor
  void g();
}
```

  - มีประโยชน์ในการแยกอินเทอร์เฟซสำหรับผู้ใช้กับผู้พัฒนาแยกกัน

จาก C เป็น C++ : Namespace

- ไลบรารีมาตรฐาน C++ ใช้ namespace **std**

```
#include <iostream>

int main()
{
   std::cout << "Hello, world!" << std::endl;
   return 0;
}
```

จาก C เป็น C++ : Namespace

- ไลบรารีมาตรฐาน C++ ใช้ namespace **std**

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

int main()
{
  cout << "Hello, world!" << endl;
  return 0;
}
```

จาก C เป็น C++ : Exception Handling

- รูปแบบต่าง ๆ ในการจัดการ error ในภาษา C :
  - ฟังก์ชัน return 0 = สำเร็จ มิฉะนั้นก็เป็น error code
  - ฟังก์ชัน return **`TRUE`** = สำเร็จ, **`FALSE`** = error
  - ฟังก์ชัน return ค่าที่ใช้งานได้ = สำเร็จ, ค่าพิเศษ = error
  - ฟังก์ชัน return 0 = สำเร็จ, ค่าไม่ใช่ 0 = error
    แล้วให้อ่านรหัส error จากค่า **`errno`** เอา
- ผู้เรียกต้องตรวจสอบค่า return ด้วย **`if`** และจัดการอย่างเหมาะสม

จาก C เป็น C++ : Exception Handling

- วิธีของ C++ : ใช้ **try-catch** และ **throw**
- ตัวฟังก์ชันที่เกิด error: **throw** error

```
const int DivideByZero = 10;

double divide (double x, double y)
{
  if (y == 0) throw DivideByZero;
  return x / y;
}
```

จาก C เป็น C++ : Exception Handling

- ผู้เรียก: **try** เรียกการกระทำ และ **catch** เพื่อจับ error

```
try {
   u = divide (10, 0);
} catch (int i) {
   if (i == DivideByZero) {
      cerr << "Error: Divide by zero." << endl;
   }
}
```

จาก C เป็น C++ : Exception Handling

- สิ่งที่ **throw** สามารถเป็นชนิดใดก็ได้
- แต่ต้อง **catch** ตามชนิดนั้น ๆ ด้วย
- โดยทั่วไปจะกำหนดชนิด exception ขึ้นมาต่างหาก

```
class DivideByZero {};
class TooSmallDivisor {};

double divide (double x, double y)
{
  if (y == 0) throw DivideByZero();
  if (y < 1.0e-20) throw TooSmallDivisor();
  return x / y;
}
```

จาก C เป็น C++ : Exception Handling

```
try {
   u = divide (10, 0);
} catch (DivideByzero e) {
   cerr << "Error: Divide by zero." << endl;
} catch (TooSmallDivisor e) {
   cerr << "Error: Too small divisor." << endl;
} catch (...) {
   cerr << "Error: General error. << endl;
}
```

- **try** หนึ่ง ๆ สามารถมี **catch** ได้หลายบล็อค เพื่อจับ exception ชนิดต่าง ๆ
- ใช้ **catch(...)** เพื่อจับ exception ที่ยังไม่มีการจับ

จาก C เป็น C++ : Exception Handling

- exception ไม่จำเป็นต้องจับทันทีที่ return จากฟังก์ชัน
- หากมี exception ที่ไม่ถูกจับเกิดขึ้น
  → ออกจากฟังก์ชันปัจจุบัน กลับไปยังฟังก์ชันชั้นบน
- ฟังก์ชันชั้นบนอาจจับ exception แทนได้ (ด้วย `try-catch`)
  → สามารถจัดการ exception ในจุดที่เหมาะสมได้
- หากไม่มีฟังก์ชันชั้นบนใดจับ exception นั้นเลยจนถึง `main()`
  → จบโปรแกรมกลางคันแบบ error

จาก C เป็น C++ : New Paradigms

- การเขียนโปรแกรมเชิงวัตถุ (Object-Oriented Programming)
  - `class`
  - operator overloading
  - inheritance
  - polymorphism
  - Run-Time Type Information (RTTI)
  - การ cast type แบบละเอียด
- Generic Programming
  - `template`

โปรดติดตามตอนต่อไป!