COMP2012 Object-Oriented Programming and Data Structures

Topic 1: Revision Example, Pointer, Reference & Const-ness

Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

Why Take This Course?

You have taken COMP1021/1022P/1022Q and COMP2011. So you can program already, right?

- Think about this: You have been learning English for many years, but can you write a novel?
- You basically have learned the C part of C++ in COMP2011 with a brief introduction to C++ classes, and you can write small C++ programs.
- But what if you are to write a large program, probably with a team of programmers?

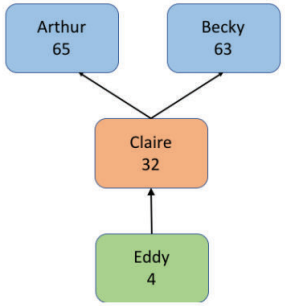
In this course, you will learn the essence of OOP with some new C++ constructs with an aim to write large softwares.

Part I

A Revision Example: Person and Family

A Revision Example: Person & Family

- It consists of a the class Person, from which families are built.
- A person, in general, has at most 1 child, and his/her father and mother may or may not be known.
- The information of his/her family includes him/her and his parents and grandparents from both of his/her parents.



Revision Example: Expected Output

```
Name: Arthur
Father: unknown
Mother: unknown
Grand Fathers: unknown, unknown
Grand Mothers: unknown, unknown

Name: Becky
Father: unknown
Mother: unknown
Grand Fathers: unknown, unknown
Grand Mothers: unknown, unknown

Name: Claire
Father: Arthur
Mother: Becky
Grand Fathers: unknown, unknown
Grand Mothers: unknown, unknown

Name: Eddy
Father: unknown
Mother: Claire
Grand Fathers: unknown, Arthur
Grand Mothers: unknown, Becky
```

Revision Example: Person Class — Header File

```
#include <iostream>     /* File: person.h */
using namespace std;

class Person
{
  private:
    char* _name;
    int _age;
    Person *_father, *_mother, *_child;

  public:
    Person(const char* my_name, int my_age, Person* my_father = nullptr,
           Person* my_mother = nullptr, Person* my_child = nullptr);
    ~Person();

    Person* father() const;
    Person* mother() const;
    Person* child() const;
    void print_age() const;
    void print_name() const;
    void print_family() const;

    void have_child(Person* baby);
};
```


Revision Example: Person Class — Implementation File I

```
#include "person.h"     /* File: person.cpp */
#include <cstring>

Person::Person(const char* my_name, int my_age, Person* my_father,
               Person* my_mother, Person* my_child)
{
    _name = new char [strlen(my_name)+1];
    strcpy(_name, my_name);
    _age = my_age;
    _father = my_father;
    _mother = my_mother;
    _child = my_child;
};

Person::~Person() { delete [] _name; }

Person* Person::father() const { return _father; }

Person* Person::mother() const { return _mother; }

Person* Person::child() const { return _child; }

void Person::have_child(Person* baby) { _child = baby; }

void Person::print_age() const { cout << _age; }
```


Revision Example: Person Class — Implementation File II

```
void Person::print_name() const
{
    cout << (_name ? _name : "unknown");
}



// Helper function
void print_parent(Person* parent)
{
    if (parent)
        parent->print_name();
    else
        cout << "unknown";
}
```

Revision Example: Person Class — Implementation File III

```
void Person::print_family() const
{
    Person *f_grandfather = nullptr, *f_grandmother = nullptr,
           *m_grandfather = nullptr, *m_grandmother = nullptr;

    if (_father) {
        f_grandmother = _father->mother();
        f_grandfather = _father->father();
    }

    if (_mother) {
        m_grandmother = _mother->mother();
        m_grandfather = _mother->father();
    }

    cout << "Name: "; print_name(); cout << endl;
    cout << "Father: "; print_parent(_father); cout << endl;
    cout << "Mother: "; print_parent(_mother); cout << endl;

    cout << "Grand Fathers: "; print_parent(f_grandfather);
    cout << ", "; print_parent(m_grandfather); cout << endl;
    cout << "Grand Mothers: "; print_parent(f_grandmother);
    cout << ", "; print_parent(m_grandmother); cout << endl;
}
```


Revision Example: Family Building Test Program

```
#include "person.h"     /* File: family.cpp */

int main()
{
    Person arthur("Arthur", 65, nullptr, nullptr, nullptr);
    Person becky("Becky", 63, nullptr, nullptr, nullptr);
    Person claire("Claire", 32, &arthur, &becky, nullptr);
    Person eddy("Eddy", 4, nullptr, &claire, nullptr);

    arthur.have_child(&claire);
    becky.have_child(&claire);
    claire.have_child(&eddy);

    arthur.print_family(); cout << endl;
    becky.print_family();  cout << endl;
    claire.print_family(); cout << endl;
    eddy.print_family();   cout << endl;
    return 0;
}
```


Part II

Reference and Pointer

Variable, Reference Variable, Pointer Variable

```
#include <iostream> /* File: confusion.cpp */
using namespace std;

int x = 5;          // An int variable
int& xref = x;      // A reference variable: xref is an alias of x
int* xptr = &x;     // A pointer variable: xptr points to x

void xprint()
{
    cout << hex << endl; // Print numbers in hexadecimal format
    cout << "x =     " << x     << "\t\tx address = "    << &x    << endl;
    cout << "xref =  " << xref  << "\t\txref address = " << &xref << endl;
    cout << "xptr =  " << xptr  << "\txptr address = "   << &xptr << endl;
    cout << "*xptr = " << *xptr << endl;
}

int main()
{
    x += 1; xprint();
    xref += 1; xprint();
    xptr = &xref; xprint(); // Now xptr points to xref

    return 0;
}
```

Pointer vs. Reference

Reference can be thought as a special kind of pointer, but there are 3 big differences:

1. A pointer can point to nothing (`nullptr`), but a reference is always bound to an object.
2. A pointer can point to different objects at different times (through assignments). A reference is always bound to the same object.

   Assignments to a reference does not change the object it refers to but only the value of the referenced object.
3. The name of a pointer refers to the pointer object. The * or -> operators have to be used to access the object.

   The name of a reference always refers to the object. There are no special operators.

This Pointer

- Each class member function implicitly contains a pointer of its class type named "this".
- When an object calls the function, this pointer is set to point to the object.
- For example, after compilation, the member function `Person::have_child(Person* baby)` of Person will be translated to a unique global function by adding a new argument:

```
void Person::have_child(Person* this, Person* baby)
{
    this->_child = baby;
}
```

- The call, `becky.have_child(&eddy)` becomes

  `Person::have_child(&becky, &eddy)`.

Return an Object by this — complex.h

```
class Complex /* File: complex.h */
{
  private:
    float real; float imag;

  public:
    Complex(float r, float i) { real = r; imag = i; }
    void print() const { cout << "(" << real << " , " << imag << ")\n"; }

    Complex add1(const Complex& x)  // Return by value
    {
        real += x.real; imag += x.imag;
        return (*this);
    }
    Complex* add2(const Complex& x) // Return by value using pointer
    {
        real += x.real; imag += x.imag;
        return this;
    }
    Complex& add3(const Complex& x) // Return by reference
    {
        real += x.real; imag += x.imag;
        return (*this);
    }
};
```


Return an Object by this — complex-test.cpp

```
#include <iostream> /* File: complex-test.cpp */
using namespace std;
#include "complex.h"

void f(const Complex a)  { a.print(); }  // const Complex a  = u
void g(const Complex* a) { a->print(); } // const Complex* a = &u
void h(const Complex& a) { a.print(); }  // const Complex& a = u

int main()
{
    // Check the parameter passing methods
    Complex u(4, 5); f(u); g(&u); h(u);

    // Check the parameter returning methods
    Complex w(10, 10); cout << endl << endl;
    Complex x(4, 5); (x.add1(w)).print();  // Complex temp  = *this = x
    Complex y(4, 5); (y.add2(w))->print(); // Complex* temp = this = &y
    Complex z(4, 5); (z.add3(w)).print();  // Complex& temp = *this = z

    cout << endl << endl; // What is the output now?
    Complex a(4, 5); a.add1(w).add1(w).print();   a.print(); cout << endl;
    Complex b(4, 5); b.add2(w)->add2(w)->print(); b.print(); cout << endl;
    Complex c(4, 5); c.add3(w).add3(w).print();   c.print();
    return 0;
}
```

Return-by-Value and Return-by-Reference

There are 2 ways to pass parameters to a function

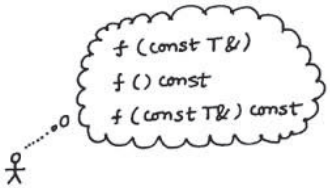
- pass-by-value (PBV)
- pass-by-reference (PBR)
  - lvalue reference: that is what you learned in the past and we'll keep just saying *reference* for lvalue reference.
  - rvalue reference (C++11)

Similarly, you may return from a function by returning an object's

- value: the function will make a separate copy of the object and return it. Changes made to the copy have no effect on the original object.
- (lvalue) reference: the object itself is passed back! Any further operations on the returned object will directly modify the original object as it is the same as the returned object.
- rvalue reference: we'll talk about this later.

Part III

const-ness



const

- const, in its simplest usage, is to express a user-defined constant — a value that can't be changed.

  `const float` PI = 3.1416;

- Some people like to write const identifiers in capital letters.
- In the old days, constants are defined by the #define preprocessor directive:

  `#define` PI 3.1416

  Question: Any shortcomings?

- const actually may be used to represent more than just numerical constants, but also const objects, pointers, and even member functions!
- The const keyword can be regarded as a safety net for programmers: If an object should not change, make it const.

Example: Constant Object of User-defined Types

```
/* File: const-object-date.h */

class Date      // There are problems with this code; what are they?
{
  private:
    int year, month, day;

  public:
    Date() { cin >> year >> month >> day; }
    Date(int y, int m, int d) { year = y; month = m; day = d; }

    void add_month() { month += 1; }; // Will be an inline function

    int difference(const Date& d)
    { /* Incomplete: write this function */ }

    void print()
    { cout << year << "/" << month << "/" << day << endl; }
};
```

Example: Constant Object of User-defined Types ..

```
#include <iostream>     /* File: const-object-date.cpp */
using namespace std;
#include "const-object-date.h"

int main()   // There are problems with this code; what are they?
{
    const Date WW2(1945, 9, 2); // World War II ending date
    Date today;
    WW2.print();
    today.print();

    // How long has it been since World War II?
    cout << "Today is " << today.difference(WW2)
         << " days after WW2" << endl;

    // What about next month?
    WW2.add_month();    // Error; do you mean today.add_month()??
    cout << today.difference(WW2) << " days by next month.\n";

    return 0;
}
```


const Member Functions

- To indicate that a class member function does not modify the class object — its data member(s), one can (and should!) place the const keyword after the argument list.

```
class Date              /* File: const-object-date2.h */
{
  private:
    int year, month, day;

  public:
    Date() { cin >> year >> month >> day; }
    Date(int y, int m, int d) { year = y; month = m; day = d; }

    void add_month() { month += 1; }; // Will be an inline function

    int difference(const Date& d) const { /* Incomplete */ }
    void print() const
    { cout << year << "/" << month << "/" << day << endl; }
};
```


const Member Functions and this Pointer

- A const object can only call const member functions of its class.
- But a non-const object can call both const and non-const member functions of its class.
- The this pointer in const member functions points to const objects. For example,
  - `int Date::difference(const Date& d) const;` is compiled to

    `int Date::difference(const Date* this, const Date& d);`
  - `void Date::print() const;` is compiled to

    `void Date::print(const Date* this);`
- Thus, the object calling const member function becomes const inside the function and cannot be modified.

const and const Pointers

- When a pointer is used, two objects are involved:
  - the pointer itself
  - the object being pointed to
- The syntax for pointers to constant objects and constant pointers can be confusing. The rule is that
  - any const to the left of the $*$ in a declaration refers to the object being pointed to.
  - any const to the right of the $*$ refers to the pointer itself.
- It can be helpful to read these declarations from right to left.

```
/* File: const-char-ptrs1.cpp */
char c = 'Y';
char *const cpc = &c;
char const* pcc;
const char* pcc2;
const char *const cpcc = &c;
char const *const cpcc2 = &c;
```

Example: const and const Pointers

```
#include <iostream>     /* File: const-char-ptrs2.cpp */
using namespace std;

int main()
{
    char s[] = "COMP2012"; // Usual initialization in the past
    char p[] {"MATH1013"}; // C++11 style of uniform initialization

    const char* pcc {s};   // Pointer to constant char
    pcc[5] = '5';          // Error!
    pcc = p;               // OK, but what does that mean?

    char *const cpc = s;   // Constant pointer
    cpc[5] = '5';          // OK
    cpc = p;               // Error!

    const char *const cpcc = s; // const pointer to const char
    cpcc[5] = '5';         // Error!
    cpcc = p;              // Error!
    return 0;
}
```


const and const Pointers ..

Having a pointer-to-const pointing to a non-const object doesn't make that object a constant!

```
/* File: const-int-ptr.cpp */
int i = 151;
i += 20;    // OK

int* pi = &i;
*pi += 20;  // OK

const int* pic = &i;
*pic += 20; // Error! Can't change i through pic

pic = pi;   // OK
*pic += 20; // Error! Can't change *pi thru pic

pi = pic;   // Error: Invalid conversion from 'const int*' to 'int*'
```


const References as Function Arguments

- There are 2 good reasons to pass an argument as a reference. What are they?
- You can (and should!) express your intention to leave a reference argument of your function unchanged by making it const.
- There are 2 advantages:

1. If you accidentally try to modify the argument in your function, the compiler will catch the error.

```
void cbr(int& x) { x += 10; }        // Fine

void cbcr(const int& x) { x += 10; } // Error!
```


const References as Function Arguments ..

2. You may pass both const and non-const arguments to a function that requires a const reference parameter.

   Conversely, you may pass only non-const arguments to a function that requires a non-const reference parameter.

```
#include <iostream>
using namespace std;
void cbr(int& a) { cout << a << endl; }
void cbcr(const int& a) { cout << a << endl; }

int main()
{
    int x {50}; const int y {100};
    // Which of the following give(s) compilation error?
    cbr(x);
    cbcr(x);
    cbr(y);
    cbcr(y);
    cbr(1234);
    cbcr(1234);
}
```

Summary: Good Practice

- Objects you don't intend to change ⇒ const objects

```
const double PI = 3.1415927;
const Date handover(1, 7, 1997);
```

- Function arguments you don't intend to change
  ⇒ const arguments

```
void print_height(const Large_Obj& LO){ cout << LO.height(): }
```

- Class member functions don't change the data members
  ⇒ const member functions

```
int Date::get_day() const { return day; }
```


Summary

- Regarding which objects can call const or non-const member functions:

Calling Object	const Member Function	non-const Member Function
const Object	√	X
non-const Object	√	√

- Regarding which objects can be passed to functions with const or non-const arguments:

Passing Object	const Function Argument	non-const Function Argument
literal constant	√	X
const Object	√	X
non-const Object	√	√

That's all!

Any questions?