

What you should have learnt

Lecture 1

Programming Methodology and Object Oriented Programming (0396A) 2002

We will start the course lightly, just to do some revision, and tell you what we will be doing in the rest of the year.

Text: *C++ Effective Object-Oriented Software Construction, 2nd Ed.* Kayshav Dattatri, Prentice Hall.

References (All from Addison Wesley)
Design Patterns. Gamma, Helm, Johnson, Vlissides.
The C++ Programming Language. Bjarne Stroustrup.

You have (at least) one whole semester experience of programming, so you already know a lot about programming...

- **What is programming:** Programming process, compiler, debugger.
- **Data in programs:** Constant, variable, expression, pointer, structure.
- **Control flow in programs:** Simple statements, conditionals, loops.
- **Programs within programs:** Functions, recursion.
- **Design techniques:** Top-down design technique, readability.
- **Proving correctness:** Invariants, pre-conditions, post-conditions.

Theoretically (and *provably*), this allows you to write any sort of programs. So **consider yourselves experts in programming.**

PMOOP(0396A)

PMOOP(0396A)-1.1

Why this course?

The problem is, what's theoretically true may not be true **in practice.**

- Every human being has only a **limited amount of time** to write programs. To write large programs, **we must collaborate.** But you don't know **how to allow collaboration.**
- Programmers can only manage a program of **limited complexity.** Once exceeded, programming becomes error-prone and inefficient.

Then we need to **partition** our program into smaller independent modules. But you don't know **how this is done**, and **how to visualize them without reconstructing the whole picture in our mind.** which would make it pointless.

- The **target** of every larger project **changes over time.** But you don't know **how to write code that can withstand changes.**

This course is there to bridge these gaps.

PMOOP(0396A)-1.2

Course information

Lecturer

Dr. Isaac K. K. To, kkto@csis.hku.hk, CYC 407
Office hour: Tue 0930–1130, Wed 1300–1500.

Tutor

Au Yeung Yip Chun, yyc@csis.hku.hk, HW 502
Consultation hour: Tue 1300–1400.
Chen Ge, gechen@csis.hku.hk, HW 519
Consultation hour: Thu 1400–1500.

Course

News: hku.csis.CSIS0396A
Web: <http://www.csis.hku.hk/~c0396a>
E-Mail: c0396a@csis.hku.hk

Schedule overview

Programming tools (3 weeks), Object-Oriented Programming concepts (4 weeks), C++ specific techniques (4 weeks), Scripting (2 weeks).

PMOOP(0396A)-1.3

Course outline

- Week 1: Introduction, separate compilation.
- Week 2: Makefile, CVS.
- Week 3: Dynamic linking.
- Week 4: Abstraction, Encapsulation and Inheritance.
- Week 5: Polymorphism.
- Week 6: Object-Oriented Design introduction.
- Week 7: Multiple inheritance and virtual base class.
- Week 8: Operator overloading.
- Week 9: Generic programming.
- Week 10: Exception handling.
- Week 11: Performance tuning.
- Week 12: Python and GUI.
- Week 13: Extending python.

PMOOP(0396A)-1.4

Course administration

Lectures

Not very formal. But be considerate: if you don't want to listen, just don't attend it. There is no roll-call, anyway.

Tutorials

On demand, usually after assignments are released.

Assessment

3-4 assignments (1 written + 2-3 written/programming mixed). 40%
One 1-hour quiz. 10%.
Exam. 50%.

Plagiarism

Will use the normal departmental policy (i.e., 1st offense—deduct half of everything except exam; 2nd offense—refer to discipline committee). No mercy.

PMOOP(0396A)-1.5

Lecture conduct

Many times students ask:

How much of the lecture should I understand in the lecture hall?

- For this course, the answer is: **everything I present**.
That's the way lectures can be interesting.
- If you have anything that you don't understand, **just ask right away**.
Don't hesitate: your problem is the problem of most of the class.
- I have an annoying tendency to forget pausing for a while after a difficult concept is presented, and thus out-pace everybody.
- If you need some more time to digest, just ask me to pause.

As long as the question is at least remotely reasonable, just ask.

PMOOP(0396A)-1.6

Pre-requisites of the course

- As an advanced programming course, **I expect you to be expert in programming**. If you failed 1117, don't bother to take this course. After all, 0396A is not a pre-requisite of many other course.
- This will be a rather **heavy course** (just like every other courses that I teach). Be sure to schedule enough time for the course (around 10 hours per week should be typical).
- For the first three weeks, the canonical reference are **manual pages** and **info pages**. Learn how to use them efficiently.
- The text and references are all very interesting. I'll put some chapters of the text at the beginning of each lecture. **Do read them**.
- The course assumes you have a **Linux** system. If you don't have it, **install one now**.

PMOOP(0396A)-1.7

Revision: aim

You should really be experts in programming. But experience tells that

- Knowing how to code does not imply that students know the **underlying model** in which the language works. Many (most?) students just do copy-and-paste.
- There are always some **concepts** that you have **missed** during the lecture last year, or **forgotten** when new concepts comes.
- The **vocabulary** of different teachers are always different, in little but important ways that understanding is adversely affected.
- Students usually don't reveal the **combined power** of concepts that they learn one-by-one.

A review solves all these problems. When you listen, think about all **discrepancies** of the concepts with your experience, and **ask**.

PMOOP(0396A)-1.8

The environment of your program

- The computer has some **memory**, a **CPU**, and some capability to perform **I/O** from input, output and storage devices.
- Our program interact with the computer through the **operating system**. It allows us to think that only our program is executing in the computer.
In reality many programs run concurrently, sharing the CPU time.
- Memory is used to hold primarily **code** and **data** of the program.
And disk cache, video RAM, etc., but typically they don't appear in our code.
- It is organised as an **array of cells**, each capable of storing a **number**. Each memory cell has an **address** to identify it.
- At any time, **exactly one point** of our program is being executed. The CPU has a small memory (register) storing the address of that point, called the **Instruction Pointer**.
CPU also has many other registers, for other purposes.

PMOOP(0396A)-1.9

Constants and bits representing them

- In our computer, memory cells are 8-bit wide **bytes**. This means each memory cell can store **8 bits**, each just a zero or one.
- There are $2^8 = 256$ ways to assign different combinations of zeros and ones into a byte, so a byte can be interpreted as a number between 0 to 255 (**unsigned char**), or -127 to 128 (**char**).
- **char** is also used to represent characters.
- Larger numbers can be represented by **combining multiple bytes**.
E.g., 2 bytes form **short**, 4 bytes form **int**, 8 bytes form **long long**.
- Real numbers are represented by **floating-point numbers**, something like the scientific notation but in binary.
but the exact representation in unimportant.

Key point: all values are just sequences of bytes.

PMOOP(0396A)-1.10

A compiled C++ program

- A compiled program is composed of **functions** and **global variables**. In the memory, there is (should be) **exactly one copy** of each **function** and each **global variable** in the program.
- These *memory* are said to be **statically allocated**.
We also speak of functions and variables being statically allocated.
- "Static" means the memory is **allocated** (get fixed address) when the program starts; and will be **destroyed** only when the program exits.
- After static allocations, a function called *main* is **called**, with at least two arguments, traditionally called *argc* and *argv*. They store the command line arguments passed to the program when it is executed.
- The program finishes when the *main* function returns.
A program can also terminate by **calling the *exit()* function** or **receiving some signal** (e.g., Control-C, segmentation fault, etc).

PMOOP(0396A)-1.11

Example

```
#include <iostream>
#include <string>

using namespace std;
string name;

int main(int argc, char *argv[]) {
    cout << "What's your name?" << endl;
    cin >> name;
    cout << "Hello, " << name << endl;
    /* &name is the address holding the variable "name" */
    cout << "name @" << &name << endl;
    /* main (without ()) is the address holding the function "main", */
    /* but cout a function pointer prints "1", so let's cast it to void *. */
    cout << "main @" << (void *)main << endl;
    cout << "argc = " << argc << endl;
}
```

PMOOP(0396A)-1.12

Functions

- A function contains a sequence of **statements** and **expressions**, which are translated to **code** by the compiler.
- Values can be passed to a function using **arguments** (or **parameters**).
- When a function is called, the code of the function is executed. A function call is completed only when the function done executing, by either “falling out of the function” or by explicit return.
- When a function returns, **a value may be passed** back to the caller. A *function call is an expression* which evaluates to the value returned.
- The above implies that **a function can call other functions**. It is legal for a function to call itself, allowing **recursion**.
- To call a function, **the instruction pointer (IP) is set to the address of the function**. To allow the caller to continue execution, **the old IP must be saved somewhere**.

PMOOP(0396A)-1.13

The execution stack

- Apart from executable statements, a function can also contain **variables declaration statements** to create **local variables** for its own use.
- Most such variables are allocated **automatically**, meaning that they are destroyed automatically when the block containing the variable exits. E.g., if you have a variable in a for-loop, it is destroyed after each iteration.
- By using a **static keyword**, you can force a variable within a function to be allocated statically. (*Still remember what it means?*) It means that it is allocated at program start, destroyed at program end.
- All such variables are created in a place called the **execution stack**, or just **stack**, of the program. “Allocating automatic variables” means to push something into the stack, and “deallocating” means to pop it out.
- The stack is also used to store **temporary value**, function **arguments**, **return values** and the **saved IP** (i.e., “return address”) of function calls.

PMOOP(0396A)-1.14

Example

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

int f(int val) {
    if (val < 1)
        return 1;
    else
        return f(val-1) + f(val-2);
}

int main() {
    cout << f(3) << endl;
}
```

Execution Stack	
Address of first call to f() in f	2
Address of call to f() in main	3
Address of call to main() in libc	argv
	argc

PMOOP(0396A)-1.15

Attributes of a variable

Each variable (or function argument) has

- An **identity**. Let's have the simplified view that it is **the address of memory allocated** to it (although optimization makes it a bit incorrect). It may be allocated on register; two variables may be given the same address.
- A **type**. This tells how to **interpret** the memory that hold the variable, and what type of **operations** can be done to it.
- A **value**. The content held in the memory.
- Zero or more **names**. A static or auto variable is named by the variable declaration statement. Additional names (aliases) may be created by reference variables.

The name refers to the variable if the name is **within scope** and not **shadowed** by a “closer” variable with the same name also within scope. Scope is of course from definition until defining block ends.

PMOOP(0396A)-1.16

Example

```
float f(int &c, int d) {
    int a = c + d;
    for (int c = 0; c < d; ++c)
        a += c;
    cout << b << endl;
    cout << c << endl;
    return d;
}

int main() {
    int a = 5, b = 6;
    cout << f(b, a) << endl;
}
```

Note that scope is **static**: you know the scope at compile time. It is usually only a part of the whole **duration** of the variable
Duration: the time when the variable is allocated and not destroyed yet.

PMOOP(0396A)-1.17

Dynamic allocation

- For various reasons, sometimes we want **variables without a name**.
E.g., we **don't know how many variables are needed** at compile time, or a function needs to **allocate memory for its caller** (local variables with names would be freed upon return), etc.
- In such scenarios, we can use a "new" expression to allocate a **nameless variable**. It returns a **value with pointer type**, which is used to access the variable **indirectly**.
To access it directly requires a name, which is exactly what we avoid.
- A variable with **pointer type** can be used to hold the pointer value.
Don't confuse between the two variables!
- Such variables are allocated from the **free store**, or **heap**. They can be destroyed only by a **delete** statement.
- When we **delete** *x*, the thing **pointed to by** *x* is deleted. (Not *x* itself!)
"delete" should probably be called *delete_the_thing_pointed_to_by*.

PMOOP(0396A)-1.18

Example

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

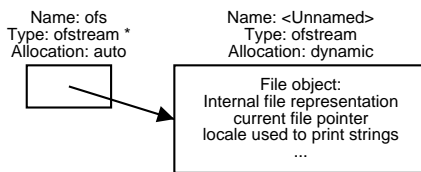
ofstream *get_file() {
    string fname;
    cout << "Filename? ";
    cin >> fname;
    return new ofstream(fname.c_str()); // Allocation for caller
}

int main() {
    ofstream *ofs = get_file(); // Pointer to access variable
    *ofs << "Hello, World!" << endl;
    delete ofs; // Not automatic
}
```

PMOOP(0396A)-1.19

Pointer versus unnamed object

Note the difference between the pointer and the unnamed object!



The pointer variable *ofs* is allocated automatically, using a variable declaration statement.

The file object is allocated dynamically using **new**. It has no name, so it can only be **accessed through a pointer** to it, e.g., by **ofs*.

Given a variable, you can get a **pointer value** to it by using the **& operator**. E.g., *&ofs* is a pointer to the variable *ofs*, of type *ofstream**.

PMOOP(0396A)-1.20

Pointers as arrays

- The C++ language supports the notion of **consecutively allocated variables** of the same type, using **arrays**.
- Given a **pointer** *x*, you can use *x + n* to get a pointer to the *n*-th variable after **x*. You can also directly get its value using *x[n]*.
- You can allocate an array in **local or global variables** using a statement like `int i[10]`. Then *i* is a pointer to integers, with 10 **int** allocated.
- You can allocate **dynamically** using an expression like `new int [10]`, which returns an **int*** typed value. To destroy that, use **delete []**.
- The **language doesn't verify the validity** of accesses (i.e., access past the end of array). The programmer must do that himself.
- Local or global multi-dimensional arrays can be created like `int i[2][3]`.
- int*** and `int[10]` are equivalent, but **int**** and `int [10][10]` are different.

PMOOP(0396A)-1.21

Structures

- At many times we want to have **several data items** allocated as a single unit, so that you can write, e.g., `TCircle c`; to allocate *c* that contains three integers for radius and *x* and *y* coordinates, and one string for a name.
- This means a **new type** called *TCircle* must be created, e.g.:

```
struct TCircle {
    int x, y, r;
    string name;
};
```

- Note that *TCircle* is **not a variable**: it is a **new type**. *x* and *y* are not variables either: they are **members** of *TCircle*.
- The compiled program contains nothing about *TCircle*.
It contains only statically allocated variables and functions.
- You can use *c.x* to access a **member** of the structure-typed variable. If you have a pointer to it instead, use `→` instead.

PMOOP(0396A)-1.22

Example: linked list

```
struct TListNode {
    int data;
    TListNode *next;
}

int main() {
    TListNode *head = 0;
    for (int i = 0; i < 10; ++i) {
        TListNode *temp = new TListNode;
        temp->data = i;
        temp->next = head;
        head = temp;
    }
    ...
}
```

PMOOP(0396A)-1.23