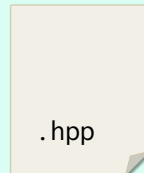


CS205

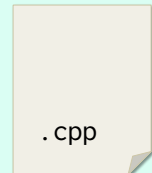
C / C++

Stéphane Faroult
faroult@sustc.edu.cn

Wang Wei (Vivian) vivian2017@aliyun.com



Interface



Implementation

C++ classes again

```
#ifndef MATRICES_HPP
#define MATRICES_HPP
class matrix {
private:
    short rows;
    short cols;
    double *cells;
public:
    matrix(int r, int c);
    ~matrix();
    matrix *matrix_add(matrix *m);
    matrix *matrix_scalar(double lambda);
    matrix *matrix_mult(matrix *m);
    matrix *matrix_inv();
    double matrix_det();
};
#endif // ifndef MATRICES_HPP
```

We have seen a sample class, but it doesn't respect a number of rules that are considered "good practices"

can be implicit (pointing to `private:`)

constructor (pointing to `matrix(int r, int c);`)

destructor (pointing to `~matrix();`)

Attributes should never be public

One rule that is more than a "good practice" is that attributes should NEVER be public. Object-oriented programming is all about "encapsulation", which simply means the privacy of attributes (and possibly some methods). A public attribute is a capital sin in object-oriented programming. It's the same in C++ as in Java or any language with classes that supports encapsulation (some languages such as Python have encapsulation of a sort, Javascript is rather murky in this respect).

Writing better classes

These are some rules frequently applied that help make the code more readable.

Capitalize class names

Method names start with lower case

Except constructors/destructors

Give special names to members

`int _val;`

or `int m_val;` *(probably better)*

NEVER start a name with two underscores

```
class Sample {
    int val;
public:
    Sample();
    ~Sample();
    void setval(int val);
};
```

```
void Sample::setval(int val) {
    val = val;
}
```

None of these rules really matters for the compiler. But they simplify code. Suppose for instance that we have a method that takes a parameter with the same name as an attribute.

We can't write this, which doesn't make any sense for the compiler

```
class Sample {
    int val;
public:
    Sample();
    ~Sample();
    void setval(int val);
};
```

```
void Sample::setval(int val) {
    this->val = val;
}
```

We can use "this", implicit pointer to the current object.

```
void Sample::setval(int val) {
    Sample::val = val;
}
```

Better, we can use the "scope operator"

```
void Sample::setval(int v) {
    val = v;
}
```

Nothing says that implementation and prototype should be strictly identical.

```
class Sample {
    int m_val;
public:
    Sample();
    ~Sample();
    void setval(int val);
};
```

```
void Sample::setval(int val) {
    m_val = val;
}
```

Having a special name for members removes any ambiguity in what is probably the easiest and simplest way.

```
class Sample {
    int m_val;
public:
    Sample();
    Sample(int val):m_val(val) {};
    ~Sample();
    void setval(int val);
};
```

Note that C++ supports a special syntax in which you can supply after the name of a constructor the name of an attribute called like a function; it means that it is initialized with this value. If initialization is the only thing that the constructor does, the body may be empty and no other implementation is needed.

VERY IMPORTANT

Object creation/destruction

It's really important to understand in C++ when and how objects are created/destroyed, because not understanding the rules can lead to unexpected crashes. First, a lot of functions (constructors, destructors) are automatically provided by C++ unless you supply one, but they aren't always suitable.

```
#include <iostream>
```

obj.cpp

```
using namespace std;
```

```
class ObjectType {
private:
    string _name;
```

I'm creating a really basic class to illustrate it. I'm creating a constructor that takes a name ...

```
public:

    ObjectType(string name) {
        _name = name;
        cout << "Creating object " << _name << endl;
    }
```

... a second constructor that takes no parameter and creates an unnamed object, an a destructor. All of them display a message when called.

obj.cpp

```
ObjectType() {
    _name = "unnamed";
    cout << "Creating object " << _name << endl;
}

~ObjectType() {
    cout << "Destroying object " << _name << endl;
}

};
```

```
int main() {
    ObjectType o1;
    ObjectType *o2p = new ObjectType("o2");

    cout << "in main()" << endl;
    return 0;
}
```

obj.cpp

In the first version of my program, I'm creating an object o1 (an object variable), then a dynamically created, Java-style named object, I'm displaying a message and exit.

```
$ g++ -o obj obj.cpp
$ ./obj
Creating object unnamed
Creating object o2
in main()
Destroying object unnamed
$
```

variable object (pointing to "o2")

delete missing (pointing to "unnamed")

What you see is that the default constructor was automatically called for o1, the variable object, and the destructor is automatically called when I quit the program. For o2, for which I explicitly called the constructor, the destructor is not called. I should also call it explicitly.

```
void func(ObjectType x) {
    ObjectType y;
    cout << "in func()" << endl;
}
```

obj.cpp

```
int main() {
    ObjectType o1;
    ObjectType *o2p = new ObjectType("o2");

    cout << "in main()" << endl;
    func(o1);
    return 0;
}
```

Let's add a call to a function that takes an object parameter (NOT an object pointer) and declares a local object variable.

```

$ g++ -o obj obj.cpp
$ ./obj
Creating object unnamed ←
Creating object o2
in main()
Creating object unnamed ←
in func()
Destroying object unnamed ←
Destroying object unnamed ←
Destroying object unnamed ←
$

```

I see two unnamed objects created ... and three destroyed, and the program doesn't crash! What is happening?

```

$ g++ -o obj obj.cpp
$ ./obj
Creating object unnamed
Creating object o2
in main()
Creating object unnamed
in func()
Destroying object unnamed
Destroying object unnamed
Destroying object unnamed

```

The additional object that is destroyed is the parameter, for which none of my constructors is called. The object is created by a "copy constructor" generated by C++, used to copy o1 into the stack. The object is freed when we return from the function.

Complex classes need to respect some rules

The compiler may provide a lot of automatic methods (constructor, destructor, copy) but there aren't always appropriate.

When classes become complex, they should be based on Coplien's canonical class.

Coplien's Canonical Class

Jim Coplien is an academic/consultant very involved in software patterns, the AGILE methodology and so forth.

```

class T {
public:
    T(); // Default Constructor
    T(const T&); // Copy Constructor
    ~T(); // Destructor (may be virtual)
    T &operator=(const T&);
        // Assignment operator

```



Don't forget the semi colon after the class definition, that must be the only place were a semi colon follows a closing curly bracket.



Coplien's Canonical Class

Why a default constructor?

```
class T {
public:
    T(); // Default Constructor
    T(const T&); // Copy Constructor
    ~T(); // Destructor (may be virtual)
    T &operator=(const T&);
        // Assignment operator
};
```

As an example, let's create a class that implements a special data type that exists in the Oracle database management system (extract from the Oracle docs below) . We'll have a "year precision" but we won't do anything with it, at it's irrelevant to the current purpose.

INTERVAL YEAR [(year_precision)] TO MONTH

Stores a period of time in years and months, where year_precision is the number of digits in the YEAR datetime field. Accepted values are 0 to 9. The default is 2. The size is fixed at 5 bytes.

YearToMonth.hpp

```
#ifndef YEARTOMONTH_HPP
#define YEARTOMONTH_HPP

class YearToMonth {
    short m_years;
    short m_months;

public:
    YearToMonth(short years);
    YearToMonth(short years, short months);
};

#endif
```

Let's create two constructors, which are easily distinguished by their number of parameters.

YearToMonth.cpp

```
#include <iostream>
#include "YearToMonth.hpp"
using namespace std;

YearToMonth::YearToMonth(short years) {
    m_years = years;
    m_months = 0;
}

YearToMonth::YearToMonth(short years, short months) {
    m_years = years;
    m_months = months;
}
```

Implementation is straightforward.

```
#include <iostream>
#include "YearToMonth.hpp"
using namespace std;

int main() {
    YearToMonth ytm1(1,7);
    YearToMonth ytm2(0,6);
    cout << "Objects created" << endl;
    return 0;
}
```

test1.cpp

A small test program
compiles and runs
nicely.

```
$ ./test1
Objects created
$
```

```
#include <iostream>
#include "YearToMonth.hpp"
using namespace std;

int main() {
    YearToMonth ytm1(1,7);
    YearToMonth ytm2(0,6);
    YearToMonth ytm_array[10];
    cout << "Objects created" << endl;
    return 0;
}
```

test2.cpp

Now let's suppose
that we'll need quite
a number of
YearToMonth
objects and want to
create an array.

```
$ g++ -o test2 test2.cpp YearToMonth.o
test2.cpp:10:17: error: no matching constructor for initialization of
'YearToMonth [10]'
    YearToMonth ytm_array[10];
    ^
./YearToMonth.hpp:10:6: note: candidate constructor not viable: requires
single
argument 'years', but no arguments were provided
    YearToMonth(short years);
    ^
./YearToMonth.hpp:11:6: note: candidate constructor not viable: requires 2
arguments, but 0 were provided
    YearToMonth(short years, short months);
    ^
./YearToMonth.hpp:4:7: note: candidate constructor (the implicit copy
constructor) not viable: requires 1 argument, but 0 were provided
class YearToMonth {
    ^
1 error generated.
$
```

Epic failure

The reason is multiple:

- 1) If you define no constructor, the C++ compiler will create a default one, which will basically reserve a number of bytes the size of one object
- 2) As soon as you define a constructor, no default constructor will be created
- 3) You cannot specify any parameter when you create an array of objects. You cannot initialize it like a C array of structures, because attributes are private. Stuck.

Note that the requirement for having a default constructor can also be found in some Java components (Java beans), for related reasons.

Coplien's Canonical Class

Why a default constructor?

Because of arrays

```
class T {
public:
    T(); // Default Constructor
    T(const T&); // Copy Constructor
    ~T(); // Destructor (may be virtual)
    T &operator=(const T&);
    // Assignment operator
};
```

Moreover

Many libraries (including the standard C++ library) require a default constructor for creating temporary objects.

If there is no constructor at all, the compiler tries to build a default one – hazardous initialization.

Some compilers may set all the bytes to 0 (which may or may not be appropriate), some compilers may not try to initialize anything.



YearToMonth.hpp

```
#ifndef YEARTOMONTH_HPP
#define YEARTOMONTH_HPP

class YearToMonth {
    short m_years;
    short m_months;

public:
    YearToMonth(short years=0);
    YearToMonth(short years, short months);
};

#endif
```

Note that "default constructor means "Constructor that accepts no parameters"

Enough to fix the problem

It can be a constructor without any parameter, or a constructor for which all parameters have a default value (note: default values are only specified in the method prototype, not in the implementation)

Coplien's Canonical Class

Why a destructor?

Because of heap memory

```
class T {
public:
    T(); // Default Constructor
    T(const T&); // Copy Constructor
    ~T(); // Destructor (may be virtual)
    T &operator=(const T&);
    // Assignment operator
};
```

We'll talk about "virtual" in a later class

It's the same problem as with the matrix structure in C: if the object allocates heap memory in the constructor or at a later stage in its life, this will not be freed by the default destructor and will lead to memory leaks.

Coppien's Canonical Class

Why a copy constructor?

```
class T {
public:
    T(); // Default Constructor
    T(const T&); // Copy Constructor
    ~T(); // Destructor (may be virtual)
    T &operator=(const T&);
        // Assignment operator
};
```

The copy constructor is also an interesting problem, strongly linked to heap memory allocation and destructors.

Copy constructor

CREATES a new object from a previous one
(different from assignment)

Created by default (byte by byte copy)

Used when objects are passed **by value** to a function, or returned by a function

Stack

<copy of obj>
<return address>

...
a = f(obj);
...

Copy constructor

CREATES a new object from a previous one
(different from assignment)

Created by default (byte by byte copy)

Used when objects are passed **by value** to a function, or returned by a function

```
T(const T&); // Copy Constructor
```

↗ If it were passed by value there would be a chicken-and-egg problem

obj.cpp

```
ObjectType(const ObjectType& original) {
    _name = original._name + "_copy";
    cout << "Creating object " << _name << endl;
}
```

I have created a copy constructor for my ObjType class, in which I set the name to ..._copy to show how the new object was obtained.

```
$ g++ -o obj obj.cpp
$ ./obj
Creating object unnamed
Creating object o2
in main()
Creating object unnamed_copy
Creating object unnamed
in func()
Destroying object unnamed
Destroying object unnamed_copy
Destroying object unnamed
```

```
void func(ObjectType x) {
    ObjectType y;
    cout << "in func()" << endl;
}

int main() {
    ObjectType o1;
    ObjectType *o2p = new ...
    cout << "in main()" << endl;
    func(o1);
    return 0;
}
```

\$ You can see here the copy constructor called when the parameter is passed, and the destructor called for this object when we return from the function.

```
$ g++ -o obj obj.cpp
$ ./obj
Creating object unnamed
Creating object o2
in main()
Creating object unnamed_copy
Creating object unnamed
in func()
Destroying object unnamed
Destroying object unnamed_copy
Destroying object o2
Destroying object unnamed
```

```
void func(ObjectType x) {
    ObjectType y;
    cout << "in func()" << endl;
}

int main() {
    ObjectType o1;
    ObjectType *o2p = new ...
    cout << "in main()" << endl;
    func(o1);
    return 0;
}
```

Destroying object o2 *delete o2p;*

\$ To do things really well I should call **delete** for the ObjectType pointer. Then I'd have four creations and four destructions.

Copy constructor

Necessary when pointers to heap areas inside the object

If your object is a plain object, such as a YearToMonth object, there is no problem, and the default copy constructor works fine.

```
class Dummy {
    int *m_tab;
    int m_sz;
public:
    Dummy(int sz=5);
    ~Dummy();
};

Dummy::Dummy(int sz) {
    cerr << "Constructor called" << endl;
    m_sz = sz;
    m_tab = new int[sz];
}

Dummy::~Dummy() {
    cerr << "Destructor called" << endl;
    delete[] m_tab;
}
```

Let's define a class for objects that contain an array of integers, dynamically created by the constructor.

Messages will help us see better what happens.

test.cpp

```
#include <iostream>
#include "Dummy.hpp"
using namespace std;

int main() {
    Dummy dum(3);

    cout << "Object created" << endl;
    return 0;
}
```

\$./test
 Constructor called → when declaring dum
 Object created
 Destructor called → when leaving main()
 \$

test2.cpp

Now let's pass dum to a function

```
#include <iostream>
#include "Dummy.hpp"
using namespace std;

void f(Dummy dum) {
    cerr << "In function f()" << endl;
}

int main() {
    Dummy dum(10);
    f(dum);
    cerr << "Leaving main()" << endl;
    return 0;
}
```

First we create dum

test2.cpp

```
#include <iostream>
#include "Dummy.hpp"
using namespace std;

void f(Dummy dum) {
    cerr << "In function f()" << endl;
}

int main() {
    Dummy dum(10);
    f(dum);
    cerr << "Leaving main()" << endl;
    return 0;
}
```

When we call the function, the default copy constructor copies every byte, including the pointer to the array.

test2.cpp

```
#include <iostream>
#include "Dummy.hpp"
using namespace std;

void f(Dummy dum) {
    cerr << "In function f()" << endl;
}

int main() {
    Dummy dum(10);
    f(dum);
    cerr << "Leaving main()" << endl;
    return 0;
}
```

When we leave the function, the destructor is called for the copy and frees memory, including the array.

test2.cpp

```
#include <iostream>
#include "Dummy.hpp"
using namespace std;

void f(Dummy dum) {
    cerr << "In function f()" << endl;
}

int main() {
    Dummy dum(10);
    f(dum);
    cerr << "Leaving main()" << endl;
    return 0;
}
```

When we leave main(), the destructor is called for dum and attempts to free the array, which is already gone.

```
$ ./test2
Constructor called
In function f()
Destructor called
Leaving main()
Destructor called
test2(2490,0x7fff79335300) malloc: *** error for object
0x7fb56a500000: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
$
```

Freeing the copy and memory in the heap

Freeing the original

Already freed

Here is what it gives without pictures

“C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.”



Bjarne Stroustrup

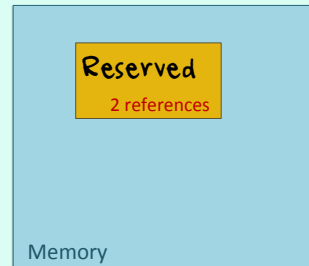
Why doesn't it happen with Java?

The garbage collector **counts** references



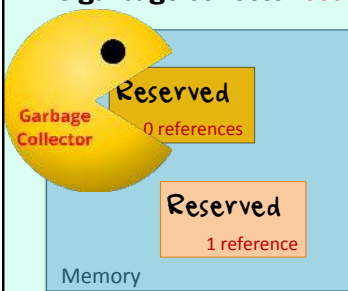
Object o = new ...
 Everytime something points to a memory area, a count is kept. When you create a new object, the count is 1.

The garbage collector **counts** references



Object o = new ...
Object o2 = o;
 If a second reference points to the same area, the count becomes 2.

The garbage collector **counts** references



Memory is only freed when the count is 0.

Object o = new ...
Object o2 = o;
o = new Object ...
o2 = null;
 If you assign a new value to o, the count will become 1, then 0 when o2 also refers to something else.

Easier on the programmer

BUT

First you can always encounter cases with references on references on references that Java will have trouble to handle.

Everything has a cost

Memory management uses some resources.

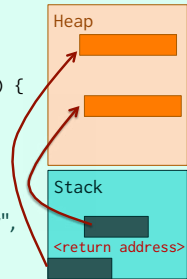
Junior developers can have many things wrong

I have seen several cases where objects were created and changed faster than the garbage collector could free memory.

And you thought
memory management in
C is complicated?

Deep copy versus Shallow copy

```
Dummy::Dummy(const Dummy& dum) {
    m_sz = dum.m_sz;
    m_tab = new int[m_sz];
    for (int i = 0; i < dum.m_sz; i++) {
        m_tab[i] = dum.m_tab[i];
    }
}
```



If we want to be safe, we need a copy constructor that performs a "deep copy", which means that it also allocates and copies anything that the original object points to.

Coplien's Canonical Class

Why a copy constructor?

```
class T { To have a destructor-safe deep copy
public:
    T(); // Default Constructor
    T(const T&); // Copy Constructor
    ~T(); // Destructor (may be virtual)
    T &operator=(const T&);
        // Assignment operator
};
```

Coplien's Canonical Class

Assignment operator?

```
class T {
public:
    T(); // Default Constructor
    T(const T&); // Copy Constructor
    ~T(); // Destructor (may be virtual)
    T &operator=(const T&);
        // Assignment operator
};
```

You can redefine operators in C++

http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B

Long and impressive list. In practice only a few operators will be overloaded.

Back to YearToMonth

```
class YearToMonth {
    short m_years;
    short m_months;

public:
    YearToMonth(short years=0);
    YearToMonth(short years, short months);
    void add(const YearToMonth ytm);
    void print();
};
```

We may want to define two methods: one to add another YearToMonth to the current one, and one to display the object.

Back to YearToMonth

```
void YearToMonth::add(const YearToMonth ytm) {
    m_years += ytm.m_years;
    m_months += ytm.m_months;
    if (m_months > 11) {
        short y = m_months / 12;
        m_years += y;
        m_months -= y * 12;
    }
}
```

Back to YearToMonth

```
void YearToMonth::print() {
    if (m_years > 0) {
        cout << m_years << " year" << (m_years > 1 ? "s" : "");
        if (m_months > 0) {
            cout << " ";
        }
    }
    if (m_months > 0) {
        cout << m_months << " month" << (m_months > 1 ? "s" : "");
    }
    cout << endl;
}
```

Back to YearToMonth

You can do more natural by overloading `+` (or `+=`) and `<<`

Our `add()` is in fact more like `+=`

operator*symbol***()**

C++ knows a lot of functions/methods named `operator<symbol>()` which can be used as such or simply as the symbol. For instance, `operator+()` allows to redefine addition.

BIG QUESTION

Function or method?

Many operators may be redefined either as functions or operators (the Wikipedia page mentioned earlier lists what the operator redefinition should look like in both cases). Which one should be used?

Redefine `<<`

Sometimes there is no choice. Let's say that we want to redefine writing an object to a stream.

```
cout << ... << object << ... << endl;
```

get a stream of things already on their way out (parameter) →
 Write more stuff ↑
 Return a stream to which more things may be added ←

Because of the way streams are used, the operator must take a stream pointer as input, write stuff to it, and return the stream pointer.

Redefine `<<`

Method?

```
class T {
public:
    ostream &operator<<(ostream &os){
        // code here ...
        return os;
    }
};
```

If we implement it as a method, it should look like this.

Redefine <<

```
T t;
```

Method?

```
t.operator<<(cout);
```

However, calling the method is equivalent to this: `t << cout;`

Not the way it is used

cout is normally on the left side. It should be an ostream method

Not possible to add a method to ostream

Redefine <<

YearToMonth.hpp

```
class YearToMonth {
    short m_years,
    short m_months;
```

Function
~~Method?~~

So it should be a function, but it should be able to access the attributes.

```
public:
```

```
    YearToMonth(short years=0);
```

```
    YearToMonth(short years, short months);
```

```
    void add(const YearToMonth ytm);
```

```
};
```

```
ostream &operator<<(ostream &os,
                    const YearToMonth &ytm);
```

Redefine <<

```
#include <iostream>
```

YearToMonth.hpp

```
class YearToMonth {
    short m_years;
    short m_months;
```

Function
~~Method?~~

Either we have methods to return each one (clumsy), hide nothing, or we allow the function to see them

```
public:
```

```
    YearToMonth(short years=0);
```

```
    YearToMonth(short years, short months);
```

```
    void add(const YearToMonth ytm);
```

```
    friend std::ostream &operator<<(std::ostream &os,
                                    const YearToMonth &ytm);
```

```
};
```

Don't use a namespace in a header file

```
using namespace std;
```

YearToMonth.cpp

```
ostream &operator<<(ostream &os,
                    const YearToMonth &ytm) {
```

```
    if (ytm.m_years > 0) {
        os << ytm.m_years << " year"
        << (ytm.m_years > 1 ? "s" : "");
```

```
        if (ytm.m_months > 0) {
            os << " ";
```

The "friend" mechanism is hated by object-orientation purists.

```
        }
        if (ytm.m_months > 0) {
            os << ytm.m_months << " month"
            << (ytm.m_months > 1 ? "s" : "");
```

```
        }
        return os;
    }
```

```
#include <iostream>                                     Test4.cpp

#include "YearToMonth.hpp"
using namespace std;

int main() {
    YearToMonth ytm1(1, 7);
    cout << "ytm1 is " << ytm1 << endl;
    return 0;
}

$ ./test4
ytm1 is 1 year 7 months
$
```

And now we can display a YearToMonth in a very natural way.

Coplien's Canonical Class

The assignment operator recommended by Coplien IS a method, because we are only dealing with objects of the class.

```
class T {
public:
    T(); // Default Constructor
    T(const T&); // Copy Constructor
    ~T(); // Destructor (may be virtual)
    T &operator=(const T&);
        // Assignment operator
};
```

But this is a method!

Function or method ?

Rule: if the current object plays the lead part **method**, otherwise **function**

Function or method ?

++ ? Method (affecting the current object)

+ ? Takes two objects, returns a third one: friend function.

Coplien's Canonical Class

```
class T {
public:
    T(); // Default Constructor
    T(const T&); // Copy Constructor
    ~T(); // Destructor (may be virtual)
    T &operator=(const T&); // Assignment operator
};
```

returns a reference for right-hand chaining $a = b = c$;

Assignment operator

Same problems as copy except that you aren't creating a new object.



Copy operator = new material

Assignment operator

Same problems as copy except that you aren't creating a new object.

Copy operator = new material

Assignment operator = old material

Possibly a need for deleting, then recreating memory areas.

As C and C++ are mostly (although not exclusively) used for system programming, we are going to be briefly acquainted with what is known as "system calls". As C is historically closely linked to Unix, most of what we'll see is related to Unix-like systems.

System Calls

Just an introduction ...
MOSTLY UNIX/LINUX

Unix/Linux manual pages

- 1 – General commands
- 2 – **System calls**
- 3 – C library functions
- 4 – File formats

Getting information about a command (or C function) is performed on a Unix system by typing "man <name>" in a console. The "manual" is divided into several sections, a full one is devoted to system calls (to show how important they are).

processes



You must always keep in mind that your program never runs alone. It's just one of many "processes" sharing hardware resources (CPU, disks, memory) and coordinated by an all-powerful set of programs, the operating system (abbreviated to OS).

Whenever your program needs a resource, it must issue a request to the OS - a "system call".



Some requests are considered to be "standard"

`fopen()`, `fread()`, `fwrite()`
`malloc()` / `new`

Others are more "advanced"

Low level I/Os
 Process management/communication

man fopen

Many standard C functions that you have been using are merely wrappers around system calls that present a more programmer-friendly interface. Check for instance what the manual says about a function such as `fopen()`

FOPEN(3) BSD Library Functions Manual FOPEN(3)

NAME
fdopen, fopen, freopen -- stream open functions

LIBRARY
 Standard C Library (libc, -lc)

SYNOPSIS
#include <stdio.h>

FILE *
fdopen(int fildes, const char *mode);

FILE *
fopen(const char *restrict filename, const char *restrict mode);

FILE *
freopen(const char *restrict filename, const char *restrict mode,
 FILE *restrict stream);

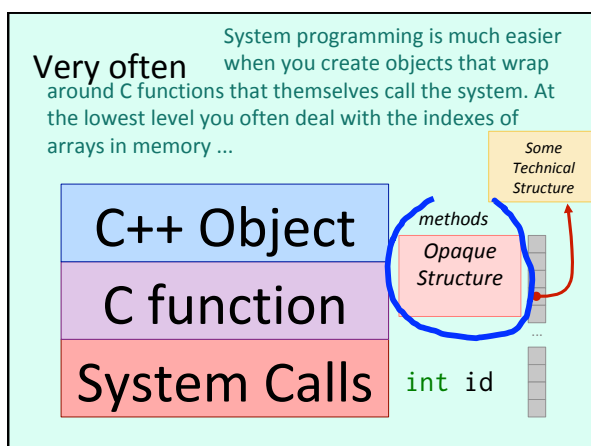
Section 3 = C functions

SEE ALSO
 open(2), close(3), fileno(3), fseek(3), funopen(3)

STANDARDS
 The **fopen()** and **freopen()** functions conform to ISO/IEC 9899:1990 (''ISO C90''). The **fdopen()** function conforms to IEEE Std 1003.1-1988 (''POSIX.1'').

BSD January 26, 2003

When you scroll to the very end, you see in the SEE ALSO section a reference to the system call `open()` which is what `fopen()` actually calls. `open()` returns an integer that identifies the stream, not an opaque `FILE *`, it takes combined numerical flags to specify the mode instead of "a", "w" or "r", but functionalities are similar.

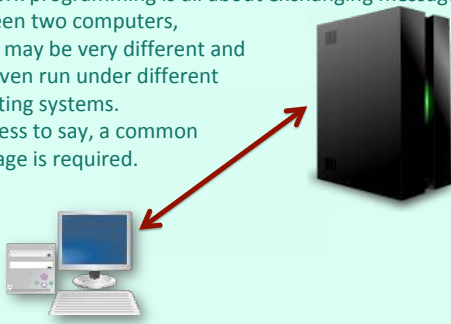


Network Programming

(a short overview)

Functions that deal with network programming in C all belong to section 2 in the manual. Most of them were initially created for a Unix-like system called "BSD" that was developed in Berkeley in the 1970s, concurrently to the official Bell Labs Unix (often referred to as "System V"). Later, good ideas from both systems were merged into newer Unix systems.

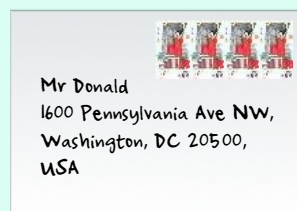
Network programming is all about exchanging messages between two computers, which may be very different and may even run under different operating systems. Needless to say, a common language is required.



Networking is very similar to snail-mail. You may want to send a message (say a nice card) and you need put it into an envelope.

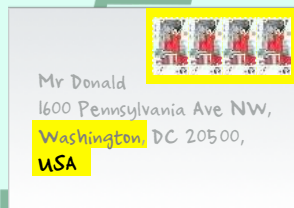


Then you need to affix a stamp (or several stamps), and write down an address, let's say in the US.



Then you can handle it to China Post.

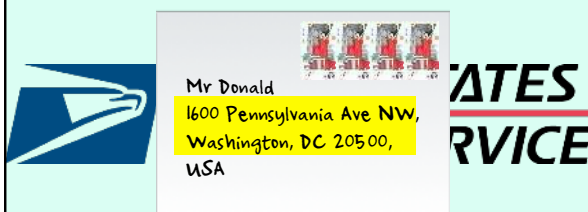
China Post are not interested in your message. They don't care about whom you are writing to, they don't care about the street nor the precise part of Washington.



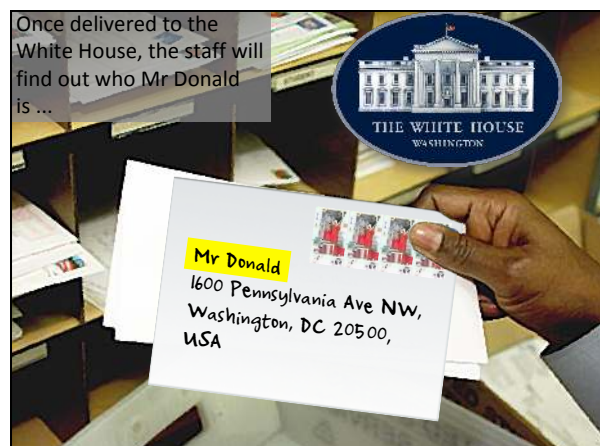
They only care about the stamps being correct, the country and, for a country as big as the US, the airport where they fly that is closest to the destination. It may be Los Angeles ...



Once there, it will be handled by USPS (United States Postal Services); they only care about the precise address in Washington.



In fact, they must first care about taking the letter to Washington, and about the precise address only when in Washington.



Several layers of information

What is interesting is that each intermediary only looks at one piece of information ...



Mr Donald
1600 Pennsylvania Ave NW,
Washington, DC 20500,
USA



Each one relevant to one go-between

Several layers of information

...then handles the message to someone else ...



Mr Donald
1600 Pennsylvania Ave NW,
Washington, DC 20500,
USA



Each one relevant to one go-between

Several layers of information

... until it reaches the addressee (or his secretaries) who reads the message (watch your words!)



Mr Donald
1600 Pennsylvania Ave NW,
Washington, DC 20500,
USA



Go-betweens only read what is relevant to them

It works exactly the same with computer networks

When you send a message over a network, it must reach a computer that may not be in the same network as you are. Your message will be sent from machine to machine, being forwarded by dedicated computers known as routers, will hop from network to network through "gateways" that are computers connected to two networks at once, and your message will be wrapped into information only used by the intermediaries.

Need for PROTOCOLS

Once again, for everybody to understand, you must follow certain sets of rules, known as a "protocol", in the same way that you are supposed to provide some special information in the address on an envelope, and to write it in a special way (not country and zipcode at the top in English) so that it's understood in any post-office.

Several protocols exist for sending messages, the one that over the years has become the most popular, the one that is used over the Internet, is known as TCP/IP.

For your application to send a message successfully, there must be a special program (generically called "listener" here) ready to receive it. With TCP/IP, this listener is attached to what is called a "port".

Application MESSAGE



What is a port? An integer value, which can be compared to a mailbox number in a block of apartments. Many machines host several programs that can receive messages and reply to them.

Port numbers are specific to a service.



Flickr: Newtown graffiti

IP address

4 bytes (32 bits)

198.51.100.97

Network address / Machine address

Mask 1111 ... / 000 ...

Talking from a machine (host) to another machine is like sending a letter, and is the job of the TCP protocol. You have an address (the "IP address") and you know the address of the other machine. Machines belong to networks also identified in the IP address.

Application Host-to-Host

TCP



16 bytes (128 bits)

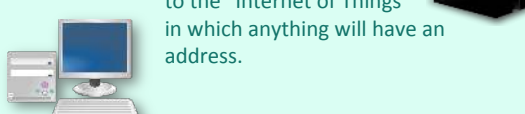
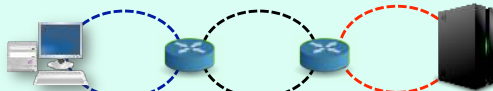
IP address 2001:0db8:85a3:0000:0000:8a2e:0370:7334

"Internet of Things"

We are running out of traditional 32-bit addresses and much larger addresses are introduced, with a view to the "Internet of Things" in which anything will have an address.

TCP



Application
Host-to-Host


"Internet" means "Inter (between, as in "international") Network" and is about transferring data from network to network. Obviously the job of the "Internet Protocol", better known as IP. It's like one postal network handling mail to another.

IP

Application
Host-to-Host
Internet

To reach another network, you must transfer via a machine on your network (Gateway) that is also connected to another network. Routers on your network will take you there.

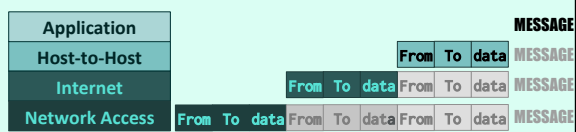


Application
Host-to-Host
Internet
Network Access

Network programming is usually presented as several independent layers that only communicate with the next one, in the same way as snail-mail. Each layer is in charge of checking some part of the information and handling the message to adjacent layers.

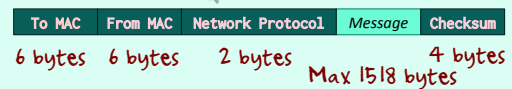
Application
Host-to-Host
Internet
Network Access

Each layer adds the information it needs, and considers the message to be what was handled to it by the preceding layer (the original message, PLUS layer specific data)



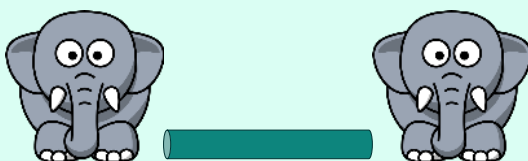
Will be modified by the various machines on the route

You sometimes have a suffix (to check that there is no transmission error)



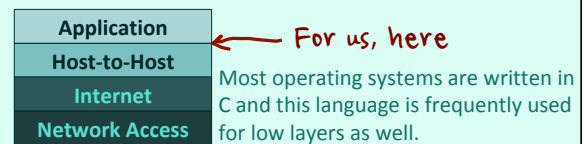
This is the lowest level message. The MAC(hine) address is a kind of serial number that identifies a network card.

Slice and dice big messages



Chunks that are sent are relatively small (1518 bytes at most). Big messages are reassembled at destination.

Where does C network programming hit?



Based on a SOCKET

stream, like a file

Some weird structures

We'll get into the details next time.