CS205

C / C++

Stéphane Faroult
faroult@sustc.edu.cn

Wang Wei (Vivian) vivian2017@aliyun.com

Pure C++

We are back to pure C++, with topics that don't exist in C such as inheritance, exceptions, and templates.

Clarification about namespaces in .hpp files

First of all, a clarification about namespaces: don't use "using namespace xxx;" in header files.

```
using namespace std;
```

in .h or .hpp

Reason

Name conflict

Methods: Class tells which one

Functions ?

What is a namespace? A grouping of names together to avoid conflicts and ambiguities. Methods can be prefixed by the name of the class they belong to avoid ambiguities. But what about functions?

Reason

```
namespace domain1 {
    // Declare functions, possibly variables
}
```

`using namespace` in included file may be dangerous

Namespaces are mostly used by people writing libraries. If you have "using namespace" in a header file, it may be included at a place where it conflicts with a very specific environment.

There is a "global namespace" just indicated by ::

```
if (::connect(sd, (struct sockaddr*)&address,
                         sizeof(address)) != 0) {
    perror("connect() failed");
    close(sd);
    return NULL;
}
```

This means "the regular C connect() function, not the method in this class"

(from tcpconnector.cpp)

tcpstream.h

```
#ifndef __tcpstream_h__
#define __tcpstream_h__

#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <string>

// using namespace std;

class TCPStream
{
  ...
```

Note that tcpstream.h contains a "using namespace". What happens if we comment it out?

```
$ g++ -c -I./tcpsockets-master -o HTTPCnx.o HTTPCnx.cpp
In file included from HTTPCnx.cpp:3:
In file included from ./HTTPCnx.hpp:5:
In file included from ./tcpsockets-master/tcpconnector.h:28:
./tcpsockets-master/tcpstream.h:37:5: error: unknown type
name 'string'; did you mean 'std::string'?
    string  m_peerIP;
    ^~~~~~
    std::string
/Applications/Xcode.app/Contents/Developer/Toolchains/
XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/iosfwd:
194:65: note:
      'std::string' declared here
typedef basic_string<char, char_traits<char>, allocator<char>
> string;
^
```

The compiler only complains about "string" and tells us what to do.

```
std::string  m_peerIP;
std::string getPeerIP();
```

All we have to do is prefix occurrences of "string" with "std::" and problems are gone.

```
using namespace std;
```

in .cpp files that are missing it

As it's no longer included though, we need to add "using namespace std;" to the .CPP files that were including tcpstream.h

Short-cut for initializing attributes

```
<constructor>: attr_name(value), ... {
}
```

Before we talk about inheritance, we should remind that you can initialize values in a constructor by giving, between the name of the constructor and its body, the name of the member attribute(s) with the value to set.

Short-cut for initalizing attributes

This was actually used in Vic Hargrave's classes:

```
TCPStream::TCPStream(int sd,
             struct sockaddr_in* address) : m_sd(sd) {

    ...
}
```

and it's exactly the same as this:

```
TCPStream::TCPStream(int sd,
             struct sockaddr_in* address) {
    m_sd = sd;
    ...
}
```

Short-cut for initalizing attributes

This is the general syntax:

```
Class::Class(parameters):attr1(val1), attr2(val2)... {

    ...
}
```

I am mentioning it now because the syntax resembles closely the one used for an important topic of the day: inheritance.

Inheritance

deriving a class from another class

Inheritance is one of the pillars of object oriented programming and refers to the ability of deriving a class from another, thus "inheriting" methods and attributes, before adding more methods and attributes. Graphical interfaces are a typical example, with all widgets having a common core.

Inheritance

The syntax for the constructor of a derived class is

```
Class::Class(parameters) : ParentClass(), attr(val)... {
    ...
}
```

Replaces super() in Java

The ParentClass constructor is invoked (with or without parameters), then you can set attribute values before the body proper.

Inheritance

It's common to talk about "inheritance" when refering to money or goods let to you by a deceased relative (or sometimes acquaintance). There is a second meaning when people say that you have inherited traits or talens from say a grandparent.

Inheritance in object oriented programming refers to the second meaning. It doesn't mean that a class is "given" attributes and methods: it's about being, not about having, and means shared DNA (so to speak).

Inheritance

Inheritance is usually applied in two cases

<*Class name*>
is implemented as a
<*Class Name*>

Purists try to restrain it to:

<*Class name*> is a <*Class Name*>

Inheritance

Implementation refers to how things are done (in Java, you talk of "interface")

<*Class name*>
is implemented as a
<*Class Name*>

All stacks allow the same basic operations, but they can be implemented in different ways.

Stack push() Array

pop() Linked list

Inheritance

There are other ways to do it (aggregates, I'll discuss them later)

<*Class name*>
is implemented as a
<*Class Name*>

```
MyStack::MyStack():LinkedStack() {
}
```

In that case, your class is implemented in a particular way.

Classical inheritance is when your new class "specializes" a more general one.

Inheritance

General

<*Class name*> is a <*Class Name*>

TravelServices
- trip_id
- price
- currency
- buy_date
- consumption_date

FlightTicket

HotelBooking

CarRental

Specialized

Inheritance

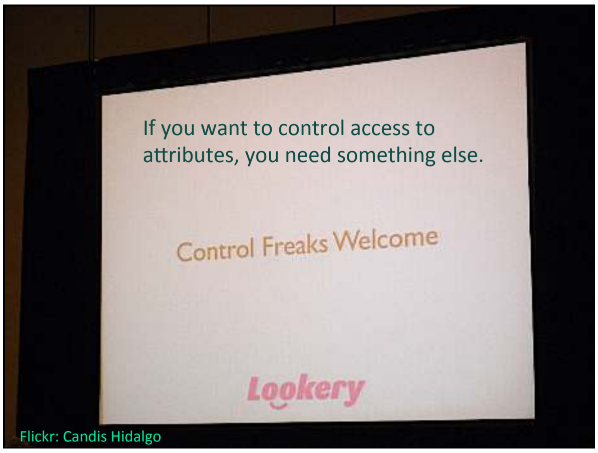
Visibility of attributes and methods?

Derived classes **don't** see what is private

Inheritance

Why? Because if a derived class could see private members, you'd just have to derive a class from a super-sensitive class to access what is private, and encapsulation would be useless and would hide nothing. Contrast it with "`friend`" functions that are declared in the class.

"You can choose your friends, but not your family"



Flickr: Candis Hidalgo

Inheritance

Of course some people complained, and a new attribute category was introduced.

```
private:
      friend
protected:
public:
```

Visible by derived classes

Inheritance

Some people advise to have all attributes "protected" when a class is supposed to be derived, but Stroustrup has a point in wanting to hide most attributes.

```
private:
      friend
protected:
public:
```

Easy on protected ...

Inheritance
Object Assignment

Rule:

Assignment only works if ALL objects on the right hand side are objects on the left hand side.

It may at times seem counter-intuitive. If a derived class D has a parent P, then D is a P and I should be able to assign a P to it? Wrong.

Inheritance

A CarRental is a TravelService, but not all TravelServices are CarRentals.

```
TravelService  s;
TravelService *sptr;
CarRental      c;

sptr = &c;
s = c;
```

Reverse FAILS

Careful!

Assignment works with pointers and references, not necessarily objects.

Inheritance

```
TravelService  s;
TravelService *sptr;
CarRental      c;

sptr = &c;

s = TravelService(c);
```

Using a Clone constructor (that takes a reference) will be safe, though.

Inheritance

Inheritance can be qualified; "public" is the most used by far.

```
class A { ... };

// public inheritance
class B : public A { ... };

// protected inheritance
class C : protected A { ... };

// private inheritance
class D : private A { ... };
```

What is public in A becomes protected in C

What is public in A becomes private in D

Inheritance Object Assignment

Rule: Only if public inheritance

assignment works only if ALL objects on the right hand side are objects on the left hand side.

There are different cases when methods have the same name, and it's important with inheritance.

Inheritance

`method overloading` Different parameters

`method overriding` This is within one class.

`method hiding`

Inheritance

`method overloading`

`method overriding` Same parameters

`method hiding` It can happen in a derived class. A method with exactly the same signature replaces the parent class method.

Inheritance

`method overloading`

`method overriding`

`method hiding` Same name

This can also happen in a derived class: a method with a similar name from the parent class will not be called, unless prefixed by the class name.

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

using namespace std;

class GeoPoint {
    private:
       string  _name;
       double  _latitude;
       double  _longitude;

    public:
       GeoPoint(string name, double lat, double lon) {
          _name = name;
          _latitude = lat;
          _longitude = lon;
       }
```

It's interesting to see what happens to methods when you combine overriding and assignment. Let's say that we have a class to store a name and geographical coordinates.

```
            void show(void) {
               cout << _name << " (" << _latitude
                    << ", " << _longitude << ")";
            }
       };
    class City: public GeoPoint {
        private:
           int _population;
        public:
           City(string name, double lat,
                double lon, int pop):
             GeoPoint(name,lat,lon), _population(pop) { }

           void show(void) {
             GeoPoint::show();
             cout << " Pop.: " << _population;
           }
    };
```

We can derive "City" from this class, add a population, and override the show() method so as to also display the population. Note how the parent method is called.

```
int main() {
    City     c = City("Singapore", 1.28333,
                      103.8333, 5610000);
    GeoPoint *g;
    c.show();
    cout << endl;
    g = &c;
    g->show();
    cout << endl;
    return 0;
}
```

In the main program, a City object is created, then displayed, then its address is assigned to a GeoPoint pointer and it's displayed again.

You can see that when the City is "downgraded" to a GeoPoint, the method that is called for displaying it is the GeoPoint method, not the City method. In Java, you would still have seen the population displayed. C++ is serious about data types ...

```
$ g++ -o GeoPoint GeoPoint.cpp
$ ./GeoPoint
Singapore (1.28333, 103.833) Pop.: 5610000
Singapore (1.28333, 103.833)
$
```

Not the same behavior as Java

Inheritance

`virtual method` May be overridden by a derived class

If you say that a method is "virtual" in a class, that means that you expect the method to be overriden (same name and parameters) in a derived class. It doesn't prevent you from implementing it in the parent class!

Abstract Class

Must be derived

An abstract class doesn't make sense in itself –- TravelService should be an abstract class, because it HAS TO BE a flight, hotel or car booking.

All constructors protected

OR

```
virtual method() = 0;
```

Initializing at least one virtual method to 0 forces the class to be abstract.

Inheritance

Multiple inheritance

```
Superman::Superman(parameters) : Bird(), Plane() {
     ...
}
```

Contrary to some other Object Oriented languages (chief among them: Java), C++ allows multiple inheritance. Multiple inheritance can quickly become complicated and it's the main reason why Gosling chose not to have it in Java.

```
class Aircraft {};
```

Let's say that we have these two base classes.

```
class Radar {};
```


```
class Awacs:
      Aircraft, Radar {};
```

We can say that an AWACS inherits from the two base classes. But it's not the only way we can look at it.

Aggregate

```
class Awacs:
      Aircraft{
   Radar rad;
};

class Awacs:
      Radar {
   Aircraft carrier;
};
```

Many people would rather advise the use of aggregates, that is having a complex object as member. You can consider the AWACS as a plane that carries a radar (with radar characteristics), or as a radar that is carried by a plane (with plane characteristics)

Aggregate

Some people could even turn it into a brand new class, that contains both a radar and a plane.

```
class Awacs {
   Radar    rad;
   Aircraft carrier;
};
```

It's very rare with modelling that there is one single "good" answer. Choice is usually dictated by current and anticipated needs, as well as personal tastes.

Is inheritance possible in pure C?

We have already seen a few lectures ago that the GNU Tool Kit (GTK), which is entirely coded in C, allows some kind of inheritance. Let's come back to it.

You can come very close

```
struct parent_t {
   ...
  };

struct child_t {
   struct parent_t parent;
   ...
  };
```

This also helps understand why inheritance and aggregates are somewhat related: you can say that a child has all the properties of the parent, plus new ones.

You can come very close

```
struct parent_t {
   ...
  };

struct child_t {
   struct parent_t parent;
   ...
  };
```

When you look in memory at a child object, if it was defined as here all the parent data will be at the beginning of the memory area.

Parent data

You can come very close

```
struct parent_t *p

struct child_t *c
```

A child pointer will point to the beginning of a memory area.

Parent data

You can come very close

```
struct parent_t *p
       = (struct parent_t *)c;
```

If you make a pointer point to the same address, it will work fine. You'll simply ignore what is specific to the child object.

Parent data

You can come very close

```
#include <stdio.h>
#include <gtk/gtk.h>

int main(int argc, char *argv[]) {
  GtkWidget *window;
  GtkWidget *label;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_title(GTK_WINDOW(window), "Example");
  gtk_window_set_default_size(GTK_WINDOW(window),
                              250, 150);
```

It works in multiple ways. gtk_window_new() says it returns the address of a GtkWidget (small) but in fact it has allocated and intitialized a GtkWindow (bigger)

Presumably some bytes are special to allow checking (signature, like files) but you can "cast" to GtkWindow without any issue.

```
  gtk_window_set_position(GTK_WINDOW(window),
                          GTK_WIN_POS_CENTER);

  label = gtk_label_new("Stop Staring at Me");
  gtk_container_add(GTK_CONTAINER(window), label);
  gtk_widget_show(label);
  gtk_widget_show(window);

  g_signal_connect_swapped(G_OBJECT(window), "destroy",
                           G_CALLBACK(gtk_main_quit),
                           NULL);

  gtk_main();

  return 0;
}
```

Data specific to a GTkContainer is probably located just after the GtkWidget data (a container is a widget with something else, a window is a container with something else), and GObject data must at the beginning of GtkWidget data.

Partial GTK object hierarchy

```
Gobject
        GInitiallyUnowned
            GtkWidget
                 GtkContainer
                     GtkBin
                         GtkWindow
```

Organizing and defining structures obviously require a lot of hard thinking, but the answer to "can it be done" is "yes, to some extent" – nobody said that it is easy.

OK, you cannot derive your own objects.

Nothing in C allows you to say that a structure derives from another structure (other than nesting them): inheritance is "by hand" and a bit rough around the edges. However, once your hierarchy is in place, you can use it in a very interesting way.

... but you can build from scratch a hierarchy of "objects"

Exceptions: Different Errors

Compile-time

Link-time

Run-time

- Detected by the application
- Detected by the library
- Detected by the Operating System / Hardware

Logic

Exceptions

Exceptions have been designed to have the library catch errors in a running program before the Operating System sees them (and delivers a lethal signal to the offending program)

Built-in Exceptions

There are many built-in exceptions in C, all deriving from std: what is called `logic_failure` is a logic error from a C programming standpoint, not from an application standpoint.

- `std::exception`
  - `std::bad_alloc`
  - `std::bad_cast`
  - `std::bad_typeid`
  - `std::bad_exception`
  - `std::logic_failure`
    - `std::domain_error`
    - `std::invalid_argument`
    - `std::length_error`
    - `std::out_of_range`
  - `std::runtime_error`

Built-in Exceptions

Runtime errors are classic stack and array management errors.

- `std::exception`
  - `std::bad_alloc`
  - `std::bad_cast`
  - `std::bad_typeid`
  - `std::bad_exception`
  - `std::logic_failure`
  - `std::runtime_error`
    - `std::overflow_error`
    - `std::range_error`
    - `std::underflow_error`

User-Defined Exceptions

Usually defined as embedded classes (classes in classes)

Avoids name conflicts

Let's now talk about your own exceptions, which are usually derived from the standard exception class (Java does the same).

```
class Outer {
     public:
        class Inner {
            Inner class declarations
        };
        Outer class declarations
};


 Outer        outerObject;
 Outer::Inner innerObject;
```

Each exception that can occur is defined as an inner class. The full exception name is qualified by the outer class name.

`std::exception`

You can follow the declaration of a method with `throw()` and the list of the exceptions that can be raised by the method. Here none of the methods is supposed to raise any exception, as it's an empty list.

```
class exception
{
  public:
    exception() throw();
    exception(const exception& rhs) throw();
    exception& operator=(const exception& rhs) throw();
    virtual ~exception() throw();
    virtual const char *what() const throw();
};
```

Exceptions are usually (it's not mandatory) derived from this class, which follows Coplien's recommendations. Method "what" allows to associate a message with the exception.

```
class MyClass {

public:
  class Error : public exception {
    public:
      virtual const char * what(void) const throw ()
      {
        return "Hopefully meaningful message";
      }
  };
  ...
};
```

Here is how you can define one exception for your class. You will create as many inner classes as things that can go wrong.

Trigger an exception with `throw`

You throw an object, not a type

```
throw Error();
```

The parentheses after `Error()` mean that we instantiate an `Error` object on the fly when we throw the exception.

To catch the exception we must give the full exception name. Optionally (with unrecoverable errors) we may want to propagate the exception "upwards" after emergency surgery, and throw it again.

```
catch (MyClass::Error &e) {
  cerr << e.what() << endl;
  // Specific processing
  throw;
}
```

Optionally propagate

GOOD PATH **BAD PATH**

Working with exceptions require having clear ideas not only about the "good path" (the program flow when everything works fine) but also the "bad path" (where and when you handle problems that can be expected or unexpected). The "bad path" is often skipped over, yet it's very important for two reasons:

1) Assessing whether the error is recoverable and if the program can continue
2) Returning enough information to developers so that they know where and why the problem happened.

Not always obvious

Where did the problem occur?

For instance, if you don't catch exceptions or simply catch and throw them again, it may be difficult to know where the problem originated. It will only pop up where you catch it. A Java stack trace may be ugly (and you certainly don't want to show something like this to an end-user) but for a developer it can be quite useful.

NEVER IGNORE THE UNEXPECTED

This should be a #1 rule when dealing with exceptions. Many developers frequently catch a number of exceptions than they know can occur and handle them, then say "whatever else happens, ignore it". This is a big mistake. At the least, any unexpected exception should be thrown. Ideally, it should be written to an error log (so that you know where it was detected), then thrown.

Exception propagation

`int main()` → calls terminate() (abort)

↑

↑

`throw`

What if a thrown exception isn't caught, or is thrown from level to level without being processed? It's always caught at the `main()` level and causes program termination. The snag is how it is done, because the default function that terminates the program is a dirty, panicky quitting.

Exception propagation

```
void farewell(void) {
  cerr << "Unhandled exception" << endl;
  cerr << "Clean program termination" << endl;
  exit(1);
}

int main(int argc, char **argv) {
  set_terminate(farewell);

  // Presumably useful code

  return 0;
}
```

You can do better by creating a function that calls the standard `exit()` function that attempts to quit cleanly (closing opened files, etc.)

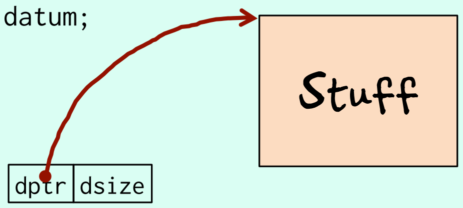
This function can be "installed" by calling `set_terminate()` at the beginning of the program.

STRONG TYPES

But let return to one of the strong points of C++: typing.

ndbm.h

```
typedef struct {
            char *dptr;
            int   dsize;
} datum;
```



The ndbm key/value store is a good example of C-style "data genericity".

```
char *
unsigned char *
void *
```

Byte address

Whatever you are calling it, `char *`, `unsigned char *` or `void *`, it's pretty much the same.

No semantic check

Debugging painful

Low productivity

Needless to say, the compiler, having no real idea about what you are trying to do, won't be able to warn you about misuse. Once your program compiles, you can brace yourself for crashes and long, painful debugging.

Here is Stroustrup's answer to these issues.

C++ Templates

C++ has also introduced a mechanism for templates, which later was "borrowed" by Java (but implemented differently)

C++ goals:

Typing is important to Stroustrup. `void *` is nightmarish in this respect.

Code once

Reuse the software component many times

Strong typing to find errors early

Overloading

```
float average(float *arr, int n) {
     float total = 0;
     for (int i = 0; i < n; i++) {
        total += arr[i];
     }
     return total / n;
}
```

Function overloading is nice but very often you need to write overloaded functions that are basically the same code. If a function takes a float argument, you can pass an integer, it will be automatically converted. However, it doesn't work with pointers, and therefore arrays. The compiler won't let you pass an int array to this function, you need to overload it with the same thing except the type of the first parameter.

```
#define THINGY   float

float average(THINGY *arr, int n) {
     float total = 0;
     for (int i = 0; i < n; i++) {
        total += arr[i];
     }
     return total / n;
}
```

One way to work around this would be to use the preprocessor (using `typedef` would also be possible), have a generic "type", and substitute whatever you need. The problem is if in the same code you need to average both an `int` and a `float` array.

Compiler with kind of preprocessor abilities

only better.

Stroustrup's answer to this was to build preprocessor-like abilities into the compiler, only better: instead of a one-off dumb substitution as the preprocessor does, the C++ compiler is able to generate on the fly several slightly different variations on a same theme.

You create a "template" with a generic class, and the compiler will use it to generate a full series of overloaded functions without your having to write anything else.

```
template<class T> float average(T *arr, int n) {
    float total = 0;
    for (int i = 0; i < n; i++) {
       total += arr[i];
    }
    return total / n;
}
```

As some people were finding "class" a bit ambiguous in this context (it also applies to plain C types, that's a difference with Java), "typename" can be used instead.

```
template<typename T> float average(T *arr, int n) {
    float total = 0;
    for (int i = 0; i < n; i++) {
       total += arr[i];
    }
    return total / n;
}
```

When you compile, the compiler will first look for a suitable function, and if none is found may be using a template.

Exact match? —Yes→ Use it

No ↓

Suitable template? —Yes→ generate the suitable function

No ↓

Compiler error

The compiler generates as many versions as needed

-> Efficient

Some langages such as Java do these types of operations dynamically, which implies runtime overhead. This isn't the case with C++.

Strong typing

Template necessarily in a header file

(not real code)

Different mechanism from Java

In fact, pointers!

Java generics only work with objects

The Java compiler adds casting

As already stated, there are significant differences with Java. Java doesn't generate new code, it just checks at compile time that everything is consistent and adds casting. One limitation is that you need to work with references, and you cannot in Java use base types in templates.

Different mechanism from Java

more flexible

Java can do a runtime check

C++ must check when compiling

faster

Java can adapt itself to varying runtime conditions, which C++ cannot. However, the compiled generated code of C++ runs faster.

WARNING

```
template <class T>
T max(T a, T b) { return (a > b ? a : b); }
```

```
int i = max(34,45);
double d = max(4.5,3.4);
```

int max(int, int)

double max(double, double)

This works fine when there is no ambiguity.

WARNING

```
template <class T>
T max(T a, T b) { return (a > b ? a : b); }
```

```
char c;
int  i,j;
...
j = max(i,c);
```

It no longer works, though, when types don't match exactly. C usually doesn't scoff at passing a char where an int is expected; conversion is automatic. It won't work with a template though.

WARNING

```
template <class T, class U>
T max(T a, U b) { return (a > b ? a : b); }
```

```
char c;
int  i,j;
...
j = max(i,c);
```

You can create a template with two parameterized classes, and this would work (however, the first parameter still has to be the same type as the returned value).

Beware that the returned type isn't a part of the signature of a function. Some people cheat by adding to a function template a dummy parameter the type of which is the type returned by the function.

What about doing it for classes as well?

Of course, the idea was extended; it's ideal for container classes, such as trees, lists or hash tables, which store data that can be of any type.

Generic tree, C-style

As already stated, plain old C doesn't really shine from a "generic" standpoint. It can do it, as long as you remain at the byte level with addresses "in memory". You end up with `void *` and `void **` pointers, the compiler has no way to check whether you aren't pointing at the right type, and if using an undefined byte address is a powerful tool it's not one to put in the hands of an inexperienced developer.

```
TSEARCH(3)              BSD Library Functions Manual              TSEARCH(3)

NAME
     tdelete, tfind, tsearch, twalk -- manipulate binary search trees

SYNOPSIS
     #include <search.h>

     void *
     tdelete(const void *restrict key, void **restrict rootp,
         int (*compar) (const void *key1, const void *key2));

     void *
     tfind(const void *key, void *const *rootp,
         int (*compar) (const void *key1, const void *key2));

     void *
     tsearch(const void *key, void **rootp,
         int (*compar) (const void *key1, const void *key2));

     void
     twalk(const void *root,
         void (*action) (const void *node, VISIT order, int level));
```

So much void that it's almost abysmal.

Node is <u>a pointer to the key of the node</u>.

Cast to a <u>double pointer</u> to the data type stored in the tree
*(for example struct myType **)*

Use <u>double indirection</u> to retrieve the original key value.

When you see how long it takes to understand simple indirection, double indirection isn't really something that in a company you trust to a young graduate.

It will probably compile.

It will probably crash when running.

and it will take an awful long time to debug.

Other option

```
typedef struct node {
                     // data here
                     struct node *left;
                     struct node *right;
                    } NODE_T;
```

... and recoding each time

```
#define DATA_STRUCT { \
                     char word[WORD_LEN]; \
                     int  counter; \
                    }

typedef struct node {
                     DATA_STRUCT
                     struct node *left;
                     struct node *right;
                    } NODE_T;
```

Tons of issues (access to members for comparison, and so forth)

A template may be a way to solve it (often, together with inheritance) if you manage to insulate tree from content management.

```
template <class T> class BinaryTree {
    private:
       struct node {
             T      m_data;
             node *m_left;
             node *m_right;
       };
       node *m_root;
       int    m_node_count;
    public:
       // ...
}
```

Beware:

Comparison can be a problem!

```
bool operator< (T &other) {
        return(data < other);
    };
};
```

You may have to redefine comparison operators for walking a tree (if you use C char arrays, you may want to redefine `operator<()` with `strcmp()` for instance)

```
struct WordCounter {
        string word;
        int    counter;
};


BinaryTree<WordCounter> word_tree;
```

And assuming that you have a suitable class at hand, a program for counting how many times a word occurs in a speech might be easy to write ...

Other example

```
template <class T>
class Stack {
   private:
      T   _s[10];
      int _sp;
   public:
      Stack() {_sp = 0;}
      void push(T x) {_s[_sp++] = x;
      T pop(void) {return _s[--_sp];}
};
```

declaration:

```
Stack<int> S;
```

or (often preferred):

```
typedef Stack<int> IntStack;
IntStack S;
```

You can also turn the stack size into a template parameter:

```
template <class T, int Max>
class Stack {
   private:
      T   _s[1+Max];
      int _sp;
      int _max;
   public:
      Stack(): _max(Max), _sp(0) {}
      void push(T x) {_s[_sp++] = x;}
      T pop(void) {return _s[--_sp];}
      int isFull(void) {return _sp == _max;}
};
```

Stack<int, 20> S;

As with function parameters, you can provide a default value:

```
template <class T, int Max = 10>
class Stack {
   private:
       T    _s[1+Max];
       int _sp;
       int _max;
   public:
       Stack(): _max(Max), _sp(0) {}
       void push(T x) {_s[_sp++] = x;}
       T pop(void) {return _s[--_sp];}
       int isFull(void) {return _sp == _max;}
};
```

Beware:

Repeat `template` each time
if separate implementation of methods

```
template <class T>
T Stack<T>::pop() {
...
}
```

Every method becomes a method template ...

Template specialization (= overriding)

```
template <class T>
class Someclass {
  // ...
}
```

In the same way you can override a parent method in a derived class, you can override a template and provide a special version.

```
template <>
class Someclass<char> {
  // redefine or extend
  ...
}
```

STL

There is a free library of templates available in C++ that has become quite important to many C++ developers and even influenced the C++ Standard Library.

Standard **T**emplate **L**ibrary

The STL is about "generic programming".

Alexander Stepanov

(1950-)

It was created by Alexander Stepanov when he was working at General Electric R&D, the Bell Labs, then Hewlett-Packard (HP)

Standard Template Library

Picture by Paul R McJones

Templates for classic data structures

- Containers
- Iterators
- Algorithms
- Functors

It covers several areas, with one which is an invented word, "functors"

Standard Template Library

- Containers
- Iterators
- Algorithms
- Functors

`vector<T,Allocator>`, `list<T, Allocator>`, `deque<T, Allocator>`

stacks and queues

set, map, hashmap

Containers are classic data structures

Standard Template Library

- Containers
- Iterators
- Algorithms
- Functors

5 types

R

W

Standard Template Library

Several types of iterators have been defined for exclusively reading data or writing it, moving forward and either reading or writing, moving in both directions or directly jumping to locations.

Containers
Iterators
Algorithms Search, Sort
Functors

Standard Template Library

Algorithms are the classic and very important searching and sorting operations.

Containers
Iterators
Algorithms
Functors also known as "function objects"

Standard Template Library

... and I'm going to illustrate "functors" and explain their usefulness.

Function call operator

`operator()`

Object callable as a function

A "functor" is simply an object that can be called as a function. The requirement is that it redefines the "function call operator", which as a method is simply called `operator()`.

Why a function object?

(instead of a regular function or method)

You might wonder what a function object has that a regular funcion or method hasn't. Spoiler: attributes.

A "functor" is simply an object that can be called as a function. The requirement is that it redefines the "function call operator", which as a method is simply called `operator()`.

```
char *simple_strtok(char *str, char sep) {
    char *p = NULL;
    static char *q;

    if (str != NULL) {
       p = str;
       q = p;
    } else p = q;
    if (p == NULL) return p;
    while (*q && (*q != sep)) q++;
    if (*q == '\0') q = NULL;
    else *q++ = '\0';
    return p;
}
```

If you want to write a simplified version of `strtok()` that takes a single char as separator, you end up with something very similar to what `strtok()` must look like, with a static pointer to remember your position in the string.

Problem:

It works well with a single string, but sometimes you need to tokenize several strings in loops.

One string at a time!

```
p = simple_strtok(str1, ' ');
while (p) {
   ...
   q = simple_strtok(str2, ' ');      RESET
   while (q) {
      ...
      q = simple_strtok(NULL, ' ');
   }
   p = simple_strtok(NULL, ' ');      LOST
}
```

Because there is a single static pointer in the function, it cannot be shared.

Solutions:

The classic C solution is to no longer have a static pointer in the function, but to pass the function a pointer on the pointer so that it can be modified.

1. Pass q as a `char **` to the function and keep one pointer per string

`char *strtok_r(...)`

`char *strsep(...)`

It's recommended to use `strsep()` rather than `strtok()`

Solutions:

2. Use a functor

The second solution is to use a functor. As every time we tokenize a new string we can instantiate a new functor, keeping the memory of where we are in the string (so far done with a static variable) can be done by an attribute that belongs to one specific functor.

```
#include <iostream>
#include <vector>

using namespace std;

class tokenizer {
   private:
    vector<char> _str;
    char         *_p;
    char         *_q;

  public:
    tokenizer(string str):_p(NULL) {
       _str = vector<char>(str.c_str(),
                           str.c_str() + str.size() + 1);
    }
```

I'm going to create a `tokenizer` class, that uses another STL template, a `vector`, which is nothing more than an array that grows automatically when needed. Method `c_str()` extracts from a C++ string the classic C \0 terminated array of chars.

And to turn my `tokenizer` class into a functor, I need to redefine `operator()` that looks very much like `simple_strtok()` except that the former static pointer is now an attribute of the class.

```
    string operator()(char sep) {
       if (_p == NULL) {
         _p = &(_str[0]);
         _q = _p;
       } else _p = _q;
       if (_p == NULL) return "";
       while (*_q && (*_q != sep)) _q++;
       if (*_q == '\0') _q = NULL;
       else *_q++ = '\0';
       return string(_p);
    }
};
```

```
int main() {
    string     tok;
    string     tok2;
    int        i = 1;
    tokenizer  next_token("2016|Mei ren yu|The Mermaid,美人鱼|cn|Stephen Chow(D),Deng Chao(A),Lin Yun(A),Luo Show(A),Zhang Yuqi(A)");
```

I'm going to parse a line which might be read from a file into I would have output the result of an SQL query. You see that I have several fields separated by '**|**' (year, title in pinyin, title in English and in Chinese, country code, credits) and that inside two of these fields I actually have comma-separated sublists. I'll first get tokens separated by '**|**', then I'll have to tokenize them again, this time using '**,**' as separator.

I have two independent functors, `next_token()` and `next_token2()`, each one keeping track of its own state and not interfering with the other.

```
   tok = next_token('|');
   while (tok.length() > 0) {
      cout << "Field " << i << endl;
      tokenizer  next_token2(tok);
      tok2 = next_token2(',');
      while (tok2.length() > 0) {
         cout << "    " << tok2 << endl;
         tok2 = next_token2(',');
      }
      tok = next_token('|');
      i++;
   }
   return 0;
}
```

```
$ ./functor_strtok
Field 1
   2016
Field 2
   Mei ren yu
Field 3
   The Mermaid
   美人鱼
Field 4
   cn
Field 5
   Stephen Chow(D)
   Deng Chao(A)
   Lin Yun(A)
   Luo Show(A)
   Zhang Yuqi(A)
$
```

When I run my program I see that I can neatly separate every field.

Now there are in computer science people that say that you should never memorize a state in a function. Those people prone what is known as "functional programming", but this is another debate ...

Beware that syntax is confusing, as when you instantiate and initialize a new object it looks a lot like calling the functor.

```
  tokenizer next_token("2016|Mei ren yu|The Mermaid,美人
鱼|cn|Stephen Chow(D),Deng Chao(A),Lin Yun(A),Luo
Show(A),Zhang Yuqi(A)");

    tok = next_token('|');
    while (tok.length() > 0) {
       cout << "Field " << i << endl;
       tokenizer next_token2(tok);
       tok2 = next_token2(',');
       while (tok2.length() > 0) {
          cout << "    " << tok2 << endl;
          tok2 = next_token2(',');
       }
       tok = next_token('|');
       i++;
    }
    return 0;
}
```

Constructors

Constructors are always preceded by the class name.

```
    tokenizer  next_token("2016|Mei ren yu|The Mermaid,美人
鱼|cn|Stephen Chow(D),Deng Chao(A),Lin Yun(A),Luo
Show(A),Zhang Yuqi(A)");

    tok = next_token('|');
    while (tok.length() > 0) {
       cout << "Field " << i << endl;
       tokenizer  next_token2(tok);
       tok2 = next_token2(',');
       while (tok2.length() > 0) {
          cout << "    " << tok2 << endl;
          tok2 = next_token2(',');
       }
       tok = next_token('|');
       i++;
    }
    return 0;
}
```

Function Calls

Function calls assign a result to a variable.

Also works – perhaps less confusing …

```
tokenizer  next_token = tokenizer("2016|Mei ren yu|The Mermaid,美人鱼|cn|Stephen Chow(D),Deng Chao(A),Lin Yun(A),Luo Show(A),Zhang Yuqi(A)");

tokenizer  next_token2 = tokenizer(tok);
```

You may find calling the constructor explicitely easier to understand.

Objects:

Harder to design well

Easier to work with

To summarize, well designing an Object Oriented program is more difficult than creating a procedural program, but when objects and methods have been set up, assembling them and using them requires much less skills. There is a trend towards two classes of programmers, those who do the hard stuff that makes life easier for others, and those who use the work of the first ones. The problem is that only people able to do the hard stuff will know what to do when there are performance issues.

C++ style cast

Lives alongside C-style casting.

static_cast<type>(*expression*)

const_cast<type>

Removes const !

C++ style cast

static_cast<type>(*expression*)

const_cast<const int>

int

Great resource

Warmly recommended.

ELLEMTEL :

Programming in C++
Rules and Recommendations

(1992)