C++ You Might Not Know

- Adam Gu, 2010/11

> C++ has indeed become too "expert friendly"
>
> - Bjarne Stroustrup, The Problem with Programming, Technology Review, Nov 2006.

C++ & C

- C++ is C with class.
- C++ is the super-set of C.

But C++'s subset C is a little different from C.

struct & union

- C struct is not a type, you have to take the keyword “struct” along, and typedef is often used to avoid this inconvenience.
- C++ struct is almost the same as class, except that the default access is public instead of private.
- C++ union can have member functions, even constructor and destructor.

Function With No Arguments

A function with no arguments in C means it takes arbitrary arguments.

Given:

```
typedef void (*foox)();
typedef void (*foo1)(int);
typedef void (*foo3)(void);
```

foo1, foo2, foo3 can be “safely” (implicitly) cast to foox, just like the cast from int*, char* to void*.

To get a function really with no arguments, use void:

```
void foo(void);
```

In C++, a function with no arguments means it just takes no argument.

*Cast From void**

> Brian W. Kernighan & Rob Pike, The Practice Of Programming, 2.6

C promotes the void* automatically:

```
int* pi = malloc(sizeof(int));
```

C++ does not; the cast is required:

```
int* pi = static_cast<int*>(malloc(sizeof(int)));
```

Constant

> In C, consts may not be used in constant expressions. This makes consts far less useful in C than in C++ and leaves C dependent on the preprocessror while C++ programmers can use properly typed and scoped consts.
>
> - Bjarne Stroustrup, The Design and Evolution Of C++, 3.8

In C++, you can use consts in constant expressions.

```
const int MAX = 4;
int a[MAX + 1];

switch (i) {
case MAX:
    ...
```

In C, you must use macro.

```
#define MAX 4
```

Forward Declarations

> Stanley Lippman, Locality of declaration, C++ Efficiency Patterns

In C, within a block, all declarations must appear before any program statements.

```
void foo() {
    int ival, *p;
    /* … */
}
```

In C++, a declaration, such as

```
int ival;
```

is itself a program statement, and so may be placed generally anywhere within the program text.

Compilation Unit

In C/C++, a source file is a compilation unit.

The simplified compilation steps:

> `preprocess → compile → link / archive`
> `source file → object file → executable / library`

Preprocess

Commands to preprocess only:

```
GCC: gcc -E
VC: cl /E or /P
```

Preprocessor processes the following things:

- macros: user defined, predefined (__cplusplus, __FILE__, etc.)
- include statement: #include
- conditional compilation: #if, #else, #ifudef, etc.
- #error, #warning

The Object

- Size of Object
- Object by Storage
- Aggregate
- Construct & Destruct Free-store Object
- RAII

Size of Object

Think about these questions before we go through them:

- Is sizeof a function?
- Do you know the values of sizeof(int), sizeof(long)...?
- Why should I use size_t?

C/C++ Data Model Notation

Here is a compact notation for describing the C language data representation on different target platforms:

> $I_{nI}L_{nL}LL_{nLL}P_{nP}$

I: int; L: long; LL: long long; P: pointer

Examples:

> `I16P32, I16L32P32, I16LP32, IL32LL64P32, ILP32LL64`

The main data models and the most popular systems using them:

short	int	long	ptr	long long	Label		Examples
...	16	...	16	...	IP16		PDP-11 Unix (1973)
16	16	32	16	...	IP16L32		PDP-11 Unix (1977); multiple instructions for long
16	16	32	32	...	I16LP32		MC68000 (1982); Apply Macintosh 68K; Microsoft operation systems (plus extras for x86 segments)
16	32	32	32	...	ILP32		IBM 370; VAX Unix; many workstation
16	32	32	32	64	ILP32LL or ILP32LL64		**Microsoft Win32;** Amdahl; Convex; 1990 Unix systems; Like IP16L32, for same reason; multiple instructions for long long
16	32	32	64	64	LLP64 or IL32LLP64 or P64	64-bit systems	**Microsoft Win64** (X64 / IA64)
16	32	64	64	64	LP64 or I32LP64		**Most Unix systems** (Linux, Solaris, DEC OSF/1 Alpha, SGI Irix, HP UX 11)
16	64	64	64	64	ILP64		HAL; logical analog of ILP32
64	64	64	64	64	SILP64		UNICOS

size_t & Those Portable Types

You might have noticed “size_t” from these functions:

```
void *malloc(size_t n);
void *memcpy(void *s1, void const *s2, size_t n);
size_t strlen(char const *s);
```

About size_t:

- size_t is the type of the result returned by sizeof operator.
- sizeof is not a function; it's a compile time operator.
- The size of size_t is chosen so that it could store the maximum size of a theoretically possible array of any type.

Why not use unsigned int?

```
void *memcpy(void *s1, void const *s2, unsigned int n);
```

This works on IP16, IP32, but not on I16LP32.

Why not use unsigned long?

```
void *memcpy(void *s1, void const *s2, unsigned long n);
```

This works but is a little less efficient.

With size_t, you simply get correctness & portability.

Portable type collection:

- size_t
- uintptr_t: the same as size_t
- ptrdiff_t: signed
- intptr_t: the same as ptrdiff_t

Data Structure Alignment

Given:

```
struct mixed_data {
    char    data1;
    short   data2;
    int     data3;
    char    data4;
};
```

What is the size of struct mixed_data? 8?

After compilation in 32-bit x86 machine:

```
struct mixed_data {
    char    data1;
    char    padding1[1];
    short   data2;
    int     data3;
    char    data4;
    char    padding2[3];
};
```

The size of struct mixed_data is 12.

Object by Storage

In C/C++, objects are classified by storage as follows:

- automatic (auto, register)
- static (static)
- free-store

The keyword auto is redundant, the following two declarations are equivalent:

```
{
  int a;
  auto int b;
}
```

In “C++ 0x”, auto lets the compiler infer the variable type from its initialization:

```
auto a = std::max(1.0, 4.0); // ‘a’ now has type double.
```

Aggregate

What's an Aggregate?

In C, array and struct are aggregates.

In C++, besides array, a class with the following conditions is aggregate:

- No user-declared constructors.
- No private or protected non-static data members.
- No base classes.
- No virtual functions.

Aggregate examples:

```
int[5];

struct person {
  std::string name;
  int age;
};

boost::array;
```

Initializing Automatic Aggregate Objects

Type	Automatic Objects

`typedef int ints_t[5];`	`ints_t ints1 = {}; // 0, 0, 0, 0, 0`
	`ints_t ints2 = { 0, 1, 2 }; // 0, 1, 2, 0, 0`
	`ints_t ints3 = { 0, 1, 2, 3, 4 }; // ...`

`struct person {`
`  std::string name;`
`  int age;`
`};`
	`person p1 = {}; // "", 0`
	`person p2 = { "john" }; // "john", 0`
	`person p3 = { "john", 26 }; // "john", 26`

`boost::array`	`boost::array<int, 3> a = { 0, 1, 2 };`

For an aggregate, default-initialization means zero-initialization.

Passing no initializer list means that the elements have an indetermined initial value.

```
boost::array<int, 4> a; // indetermined initial value.
```

Initializing Free-Store Aggregate Objects

Initializer list doesn't apply to free aggregate objects. Free aggregate objects can be zero initialized with ().

```
struct list_node {
    int value;
    list_node* next;
};

list_node* ln1 = new list_node();
list_node* ln2 = new list_node;
```

The difference between ln1 & ln2 is that, *ln1 is zero initialized while *ln2 is undeterminate.

Zero Initializing Member Aggregates

- The () following a member's name default-initializes that member.
- For aggregate member, default-initialization means zero-initialization.
- In the case of a member with a non-trivial constructor, default initialization means invoking the default constructor for that object.

How to implement the default constructor of the following class?

```
class C {
public:
    C(); // how to implement it?
private:
    struct S {
        int x;
        int* p;
        bool b;
    };
    S s;
    int d[5];
};
```

Like this?

```
C::C() {
    memset(&s, 0, sizeof(S));
    d[0] = d[1] = d[2] = d[3] = d[4] = 0;
}
```

The neat solution:

```
C::C() : s(), d() {}
```

Non-Zero Initializing Member Aggregates

Manually assign values to each member:

```
C::C() {
    s.x = 5;
    s.p = new char[s.x];
    s.b = true;
    d[0] = 9;
    ...
}
```

C++ 0x:

```
class C {
    S s { 5, new char[5], true }
    int d[5] { 9, 9, 9, 9 }
};
```

Construct & Destruct Free-Store Objects

`Mechanisms`

- malloc, realloc / free
- new operator / delete operator
- operator new / operator delete

new operator

```
[::] new [placement] new-type-name [new-initializer]
[::] new [placement] ( type-name ) [new-initializer]
```

operator new

```
// throw exception on fail
void *operator new(size_t size) throw(std::bad_alloc);
// return null on fail
void *operator new(size_t size, const std::nothrow_t&) throw();

void *operator new[](size_t size) throw(std::bad_alloc);
void *operator new[](size_t size, const std::nothrow_t&) throw();

// placement new
void *operator new(size_t, void *where) throw();
void *operator new[](size_t, void *where) throw();
```

Operator	**Scope**	
::operator new	Global	Built-in types; Class type without user-defined operator new; Arrays of any type.
class-name::operator new	Class (static)	Class type with user-defined operator new.

new operator & operator new

```
std::string* name = new std::string("Andy");
```

```
void* p = ::operator new(sizeof(std::string));
std::string* name = static_cast<std::string*>(p);
name->basic_string::basic_string("Andy");
```

```
void* operator new(...) {
    void* p;
    while ((p = malloc(size)) == 0) {
        // call new handler to get more memory or return null
pointer / throw exception
    }
    return p;
}
```

nothrow & throw

```
char* p = new (std::nothrow) char[0x7ffffffe];
if (!p) {
    // failed to allocate.
}
```

```
char* p;
try {
    p = new char[0x7ffffffe];
} catch (std::bad_alloc&) {
    // failed to allocate.
    return 1;
}
```

An example of placement new

```
const size_t count = 3;
char* buf = new char[sizeof(std::string) * count];

std::string* strs[count];
const char* str_values[count] = { "1", "2", "3" };
for (size_t i = 0; i < count; ++i) {
    strs[i] = new (buf + i * sizeof(std::string))
std::string(str_values[i]);
}

for (size_t i = 0; i < count; ++i) {
    strs[i]->~basic_string();
}
```

True or False

```
new int*;
new int&;

const int* ci = new const int(3);

int (**p) () = new (int (*[7]) ());

typedef void (*func_t)();
func_t* p = new func_t[7];

void f() {}
new f;
```

```
// C++ 0x
enum E_NUM { ONE = 1, TWO, THREE };
E_NUM* e = new E_NUM(ONE);

new class C {}; ();
new (class C {};) ();

const size_t dim = 10;
char (*pchar)[dim] = new char[10][dim];

size_t dim = 10;
char (*pchar)[dim] = new char[10][dim];
```

RAII

Resource Acquisition Is Initialization.

RAII can be used with:

- Languages which can have user-defined types allocated on the stack ("automatic" objects in C/C++ terminology) and cleaned up during normal stack cleanup (whether because of a function returning, or an exception being thrown). E.g. C++.
- Languages which have reference-counted garbage collection and hence predictable cleanup of an object for which there is only one reference. E.g. VB6 (COM, AddRef(), Release(), The CComPtr and CComQIPtr templates provided by ATL)

The design of RAII classes:

- Class that is designed purely, or at least primarily, to provide RAII semantics.
- Class that is designed to give access to a resource at an appropriate level of abstraction.

RAII in STL:

> `std::basic_ifstream`, `std::basic_ofstream` and `std::basic_fstream`
> `std::auto_ptr`

In C the following is common:

```
{
    FILE* file = fopen( ... );
    // ...
    fclose(file);
}
```

In C++ we write this:

```
{
  std::fstream file( ... );
  // do sth with file ...
} // here f gets auto magically destroyed and the destructor frees
the file
```

RAII Rationale

Stack winding & unwinding:

- When program run, each function(data, registers, program counter, etc) is mapped onto the stack as it is called. This is stack winding.
- Unwinding is the removal of the functions from the stack in the reverse order.

Unwind the stack without returning: exception, setjmp/longjmp

The exception way of unwinding:

> The process of searching “up through the stack” to find a handler for an exception is commonly called “stack unwinding.” As the call stack is unwound, the destructors for constructed local objects are invoked.
>
> - Bjarne Stroustrup, The C++ Programming Language, 14.4

The Type of Object

We won't talk about the primitive types and struct. Primitive types are simple, and struct is almost the same as class.

Union

- The elements of the union occupy the same physical space in memory;
- The size of the union is the one of the greatest element;
- The type of the object stored in a union is unknown to the complier, so it doesn’t make much sense to have member functions for union.

Union Use Case: Unify Interface

In Win32, messages pass their specific data through two 32-bit parameters: WPARAM and LPARAM:

```
LRESULT MsgProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam);
```

The user has to remember the meaning of WPARAM and LPARAM for each type of message:

```
{
  switch (msg) {
  case WM_USER:
     if (lParam == WM_RBUTTONUP) { /*...*/ }
     break;

  case WM_COMMAND:
     switch (LOWORD(wParam)) {
     case IDM_EXIT:
```

```
            SendMessage(hwnd, WM_CLOSE, 0, 0);
            break;
        case IDM_CREATE:
            create();
            break;
        //...
        }
    // ...
    }
}
```

Xlib chooses union to get type safety and convenience of use:

```
typedef union _XEvent {
    XAnyEvent xany; // the common fields
    XKeyEvent xkey;
    XButtonEvent xbutton;
    XMotionEvent xmotion;
    ...
} XEvent;
```

Each event struct might keep a type field at the beginning, e.g.,

```
struct XAnyEvent {
    int type;
    // ...
}
```

With union, the despatch of event is explicit.

```
{
  switch (evt.xany.type) {
  case <button event>:
     // evt.xbutton.<button event's specific member>
     break;

  case <key event>:
     // evt.xkey.<key event's specific member>
     //...
     }
  // ...
  }
}
```

Union Use Case: Big endian and little endian test

The Linux kernel uses the following union to test endianess:

```
static union { char c[4]; unsigned long mylong; } endian_test = { {
'l', '?', '?', 'b' } };
#define ENDIANNESS ((char)endian_test.mylong)
```

Usage:

```
if (ENDIANNESS == 'l') /* little endian */
if (ENDIANNESS == 'b') /* Big endian */
```

Class

Accessor Control

- The access modifiers are class level, not object level.

```
class foo {
   int a;
public:
   int bar(foo* f) {
      return this->a + f->a;
   }
};
```

If they’re object level, everything will be public, otherwise operations like copy contructor cannot be implemented.

Is the following code compilable?

```
class B {
protected:
    int a;
};

class D : public B {
public:
    int f(B* b) {
        return this->a + b->a; // Compilable?
    }
};
```

- The access modifiers are compile time, not runtime.

```
class Node {
public:
    Node() : value(0) {}
    int getValue() const { return value; }
private:
    int value;
};
```

```
Node node;
*reinterpret_cast<int*>(&node) = 1;
std::cout << node.getValue() << std::endl;
```

Member Function Pointers

- Member function pointers are restricted to only point to member functions of a single class.
- Member function pointers might be small structures, that encode information about a function’s virtualness, multiple inheritance and so on, rather than a single pointer (to the memory address of the start of the member function's code).
- Member function pointers cannot be dereferenced by themselves, they must be called on behave of an object.

Given:

```
class Graphic {
    void draw() {}
};
```

The pointer to member function draw():

```
void (Graphic::*draw)() = &Graphic::draw;
```

Note that you cannot emit the &. This is different from normal function pointer.

Use typedef to make it clear:

```
typedef void (Graphic::*draw_t)();
draw_t draw = &Graphic::draw;
```

To call member function pointer, you must do it on behalf of an object, which then provides the “this” pointer:

```
Graphic g;
(g.*draw)();
```

The syntax is a little confusing, the following macro helps:

```
#define CALL_MEMBER_FN(obj, mfp) ((obj).*(mfp))
```

The size of the member function pointer is not the same as normal pointer.

Given:

```
class Graphic {
public:
    virtual void draw() {}
};

class Line : public Graphic {
public:
    virtual void draw() {}
};
```

```
class Listener {
};

class OrthoLine : public Graphic, public Listener {
public:
    virtual void draw() {}
};
```

Test result in GCC and MSVC:

	g++	**MSVC (default config.)**
`sizeof(void (Line::*)())`	8	4
`sizeof(void (OrthoLine::*)())`	8	8
`sizeof(void (OrthoLine::*)())`	8	8

Cast from member function pointers among class hierarchy:

```
typedef void (Graphic::*graphic_draw_t)();
graphic_draw_t draw = static_cast<graphic_draw_t>(&Line::draw);

typedef void (Line::*line_draw_t)();
line_draw_t draw = static_cast<line_draw_t>(&Graphic::draw);
```

This mostly happens in a wxWidgets (or MFC) like GUI library.

Dereference member function pointers with NULL object.

Given:

```
class Graphic {
std::string id;
public:
    void draw() { ... }
};
```

This is OK:

```
Graphic::draw() {
  std::cout << "graphic::draw()" << std::endl;
}
```

This causes crash:

```
Graphic::draw() {
  std::cout << id << std::endl; // oops!
}
```

But this is OK:

```
Graphic::draw() {
  std::cout << &id << std::endl; // 0x00000000
}
```

An example from Linux kenel's list implementation:

```
#define list_entry(ptr, type, member) \
    ((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
```

Pointer & Reference

What's the print result of the following one-line program?

> An example from the best one liner of 1987 4th IOCCC.
>
> - David Korn

```
main() { printf(&unix["\021%six\012\0"],(unix)["have"]+"fun"-0x60);}
```

Hint: Note that the following expressions yield the same value:

```
x[3]   *(x+3)   *(3+x)   3[x]
```

More hint: see http://www0.us.ioccc.org/1987/korn.hint.

+ and - of Pointer

Given pointer to type T (T is not void):

```
T* p;
```

The statement

```
p + 1
```

is equivalent to

```
static_cast<char*>(p) + sizeof(T) // or
static_cast<uintptr_t>(p) + sizeof(T) // or
static_cast<size_t>(p) + sizeof(T)
```

The + or - of void* doesn't make sense:

```
void* p;
// ...
++p; // cannot compile!
p = p + 1;  // cannot compile!
```

The distance of two pointers can be saved in a variable of type ptrdiff_t or intptr_t:

```
int *p1, *p2;
ptrdiff_t d = p2 - p1; // or intptr_t d = ...
```

Given

```
T a, b;
```

We have

```
(&a - &b) * sizeof(T) == (ptrdiff_t)((uintptr_t)&a - (uintptr_t)&b)
```

Reference

> - Stanley Lippman, C++ Efficiency Patterns

Is this compilable?

```
void pump1(int& i) { }
void pump2(const int& i) { }

long lval = 24;
pump1(lval); // ?
pump2(lval); // ?
```

Why did C++ add references?

- In order to support operator overloaded, the existing type representations within C were inadequate.
- Bjarne Stroustrup felt that an additional type representation, that of a reference, needed to be introduced.
- A reference supports the syntax of object access while at the same time supporting the shallow intialization.

Given a class named Matrix, to support the operator + for it, the solution without reference is:

```
Matrix operator+(Matrix m1, Matrix m2);
```

You know that it has performance issue, because it passes parameters “by value”.

Using pointers elliminates the performance issue:

```
Matrix operator+(Matrix* m1, Matrix* m2);
```

But the the pointer syntax is non-intuitive and error-prone as well.

Being non-intuitive is unacceptable since the idea behind operator overloading is to make the use of the class object intuitive.

The pointer syntax cannot stop programmers to write the code like this:

```
Matrix d = &a + &b + &c; // oops!
```

It cannot compile!

The reference type provides a solution to the performance shutcoming of the object syntax:

- A reference provides the efficiency of a pointer. It avoids the overhead of the copy of large objects.
- Similarly, a reference provides the intuitive syntax of object manipulation.

Constants

> const was a useful alternative to macros for representing constants only if global consts were implicitly local to their compilation unit. Only in that case could the compiler easily deduce that their value really didn't change.
>
> - Bjarne Stroustrup, The Design and Evolution of C++, 3.8

Scope of Global Constants

Global consts are implicitly local to their compilation unit.

- const.h

```
const int CONST = 1;
```

- test_1.cc

```
#include "const.h"
const void* get_const_address_1() { return &CONST; }
```

- test_2.cc

```
#include "const.h"
const void* get_const_address_2() { return &CONST; }
```

- main.cc

```
extern const void* get_const_address_1();
extern const void* get_const_address_2();

int main() {
   assert(get_const_address_1() != get_const_address_2());
}
```

The Conversion / Cast of Type

Language Provided

- static_cast: compile-time checked conversion;
- dynamic_cast: run-time checked conversion;
- reinterpret_cast: unchecked conversion;
- const_cast: const conversion.

implicit_cast

```
template<typename To, typename From>
inline To implicit_cast(From const &f) {
  return f;
}
```

Usage:

```
double d = 3.14;
int i = 3;
std::min(d, implicit_cast<double>(i));
```

You might have two questions:

- Can I do it simply with std::min(d, i)?
- Why not just use static_cast?

The answers:

- implicit_cast is only needed in special circumstances in which the type of an expression must be exactly controlled, to avoid an overload, for example.
- The advantage of implicit_cast over other alternative casts is that someone reading the code can immediately tell that this is simply an implicit conversion, not a potentially dangerous cast (not exactly correct, see follows).

More detailed:

```
int t = d; // warning: possible loss of data
std::min(t, i);
std::min(implicit_cast<int>(d), i); // warning: possible loss of data


std::min(static_cast<int>(d), i); // no warning; dangerous
std::min((int)d, i);// no warning; dangerous
```

implicit_cast simplifies the use of implicit cast.

Another example: cast from enum to enum.

```
enum E1 { e1_0, e1_1, e1_2 };
enum E2 { e2_0, e2_1, e2_2 };

E1 e = static_cast<E1>(e2_0);
```

Some compilers don't like enum to enum casts, so we implicit_cast to int first:

```
E1 e = static_cast<E1>(implicit_cast<int>(e2_0));
```

down_cast

```
template<typename To, typename From>
inline To down_cast(From* f) {
  if (false) {
    implicit_cast<From*, To>(0);
  }

#if !defined(NDEBUG) && !defined(GOOGLE_PROTOBUF_NO_RTTI)
  assert(f == NULL || dynamic_cast<To>(f) != NULL);  // RTTI: debug
mode only!
#endif
  return static_cast<To>(f);
}
```

Inline & Template

Principles

- The compilation unit using an inline must be able to see the definition of the inline.
- The instantiation of a template must be able to see the definition of the template.
- The compilation unit using a (full) specialization of a template doesn’t have to see the definition of the specialization. To see the declaration is enough, just as non-template. (If the specialization is in header file, it is always inlined.)

Template vs. Macro

Guideline: Avoid macros when possible.

How to avoid macros? Use template (for genericity) and inline (for efficiency).

E.g., replace this kind of macros:

```
#define MIN(a, b) ((a) < (b) ? (a) : (b))
```

with:

```
template <typename T>
inline T min(const T& a, const T& b) {
    return a < b ? a : b;
}
```

You can’t avoid macros if you need the ability to paste tokens together:

This is an example from our project:

```
struct image_info {
    std::string file; // file name
    unsigned char* img2c; // the image used as default
    size_t img2c_len; // size of img2c
};
#define _PNG(name) #name ".png", name##_png, sizeof(name##_png)
```

And this is from Linux kernel (see it again :):

```
#define list_entry(ptr, type, member) \
    ((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
```

Template Specialization

Template specialization can be considered as a kind of static dispath.

NOTE: Template partial specialization won’t be mentioned here.

Specialization of function

swap:

```
template <class T>
void swap(T& a, T& b) { T tmp = a; a = b; b = tmp; }

template <class T>
void swap<vector<T> >(vector<T>& a, vector<T>& b) { a.swap(b); }
```

Template specialization & template overloading

```
template <class Base, class Exponent>
Base pow(Base b, Exponent e);

template <>
int pow(int b, int e); // specialization

template <class Base>
Base pow(Base b, int); // overloading
```

Specialization of class

Vector bool:

```
template <typename T>
class vector {
    T* vec_data;
  // ...
};

template <>
class vector <bool>
{
    unsigned int *vector_data;
  // ...
};
```

Type traits:

```
template<typename T> struct remove_pointer { typedef T type; };
template<typename T> struct remove_pointer<T*> { typedef T type; };
template<typename T> struct remove_pointer<T* const> { typedef T
type; };
```

Usage:

```
remove_pointer<int>::type i = 1;
remove_pointer<int*>::type j = 2;
```

Google chromium IPC uses template specialization to define how a data type is read, written and logged in the IPC system.

```
template <class P> struct ParamTraits {
};

template <>
struct ParamTraits<bool> {
  typedef bool param_type;
  static void Write(Message* m, const param_type& p);
  static bool Read(const Message* m, void** iter, param_type* r);
  static void Log(const param_type& p, std::string* l);
};

template <>
```

```
struct ParamTraits<int> {
  typedef int param_type;
  static void Write(Message* m, const param_type& p);
  static bool Read(const Message* m, void** iter, param_type* r);
  static void Log(const param_type& p, std::string* l);
};

template <>
struct ParamTraits<std::string> {
  typedef std::string param_type;
  static void Write(Message* m, const param_type& p);
  static bool Read(const Message* m, void** iter, param_type* r);
  static void Log(const param_type& p, std::string* l);
};
```

Exercises

Smart pointer like std::auto_ptr or boost::scoped_ptr.	
A function to check if a given string is a valid email address. `bool is_valid_email_address(const char* s);`	
Dynamic array in C.	
Dynamic array in C++ with template.	
List in C.	
List in C++ with template.	
Heap sort in C.	
Heap sort in C++ with template.	

Quick sort in C.	
Quick sort in C++ with template.	
String class with: • `copy constructor & operator =` • `operator []` • `operator <<` • `operators <, >, ==, etc.` • `trim & substr`	
Add reference couting to the above class string to support copy-on-write.	