Advanced Topics on C++

Zoltán Porkoláb, PhD.

Outline

- Constant magic
- Compiling, linking
- Constructors, destructors
- Pointers and references
- Advanced memory usage
- Smart pointers
- Advanced template techniques
- Template metaprogramming

Constant magic

- Constants in C++
- Const correctness
- Pointers to const
- Const member functions
- Const and mutable members
- STL const safety
- Constexpr

Design goals of C++

- Type safety
- Resource safety
- Performance control
- Predictability
- Readability
- Learnability

Mapping semantic issues to syntax

```
/* C language I/O */

#include <stdio.h>
int main()
{
   FILE *fp = fopen( "input.txt", "r");

   . . .

   fprintf( fp, "%s\n", "Hello input!");
   return 0;   /* not strictly necessary since C99 */
}
```

- Runtime error!

Mapping semantic issues to syntax

```
/* C++ language I/O */

#include <iostream>
#include <fstream>
int main()
{
   std::ifstream f;

   . . .

   f << "Hello input!" << std::endl;
   return 0;
}
```

- Compile-time error!

Mapping semantic issues to syntax 2.

```
#include <stdio.h>
int main()
{
    int fahr;
    for ( fahr = -100; fahr <= 400; fahr += 25 )
    {
        printf( "Fahr = %d,\tCels = %d\n", fahr, 5./9.*(fahr-32) );
    }
    return 0;
}

$ ./a.out
Fahr = -100,Cels = 913552376
Fahr = -75, Cels = -722576928
Fahr = -50, Cels = -722576928
Fahr = -25, Cels = -722576928
Fahr = 0,   Cels = -722576928
Fahr = 25,  Cels = -722576928
```

Mapping semantic issues to syntax 2.

```
#include <iostream>
int main()
{
    for ( int fahr = -100; fahr <= 400; fahr += 25 )
    {
        std::cout << “Fahr = “ << fahr <<
                          “, Cels = “ << 5./9.*(fahr – 32) << std::endl;
    }
    return 0;
}
```

```
$ ./a.out
Fahr = -100,Cels = -73.333333
Fahr = -75, Cels = -59.444444
Fahr = -50, Cels = -45.555556
Fahr = -25, Cels = -31.666667
Fahr = 0,   Cels = -17.777778
Fahr = 25,  Cels = -3.888889
```

Const values in C++

- Const literals, strings
- Preprocessor defined values
- Named consts
- Const members in class
- Const methods
- Constexpr from C++11/14

Const literals

```
char t1[] = {'H','e','l','l','o','\0'};
char t2[] = "Hello";
char t3[] = "Hello";

char  *s1 = "Hello"; // s1 points to 'H'
char  *s2 = "Hello"; // and s2 points to the same place

// assignment to array elements:
*t1 = 'x'; *t2 = 'q'; *t3 = 'y';

// modifying string literal: can be segmentation error:
*s1 = 'w'; *s2 = 'z';
```

Preprocessor macro names

```
#define LOWER  -100
#define UPPER   400
#define STEP     40

int main()
{
  for( int fahr = LOWER; UPPER >= fahr; fahr += STEP )
  {
    std::cout << "fahr = " << std::setw(4) << fahr
          << ", cels = " << std::fixed << std::setw(7)
          << std::setprecision(2) << 5./9. * (fahr-32)
          << std::endl;
  }
  return 0;
}
```

Named constants

```
// definition of non-extern const
// visible only in local source file
const int ic = 10;


// definition of extern const
// has linkage: visible outside the source file
extern const int ec = 30;


// declaration of extern const
// has linkage: visible outside the source file
extern const int ec;
```

Named constants know more

```
int f(int i) { return i; }
int main()
{
  const int c1 = 1;    // initialized compile time
  const int c2 = 2;    // initialized compile time
  const int c3 = f(3); // f() is not constexpr
  int t1[c1];
  int t2[c2];
  // int t3[c3]; // ISO C++ forbids variable-size array
  switch(i)
  {
  case c1: std::cout << "c1"; break;
  case c2: std::cout << "c2"; break;
  // case label does not reduce to an integer constant
  // case c3: std::cout << "c3"; break;
  }
  return 0;
}
```

When named const needs memory?

```
int f(int i) { return i; }

int main()
{
  const int c1 = 1;    // initialized compile time
  const int c2 = 2;    // initialized compile time
  const int c3 = f(3); // f() is not constexpr
  const int *ptr = &c2;
  return 0;
}
```

- Named pointers require storage when
  - Initialized at run-time
  - Address is used

Be careful with optimization!

```
int main()
{
    const int ci = 10;
    int *ip = const_cast<int*>(&ci);
    ++*ip;
    cout << ci << " " << *ip << endl;
    return 0;
}
```

```
$ ./a.out
10 11
```

Const correctness

```
#include <iostream>

int my_strlen(char *s)
{
    char *p = s;
    while ( ! (*p = 0) ) ++p;
    return p – s;
}

// This program likely cause run-time error
int main()
{
    char t[] = "Hello";
    std::cout << my_strlen(t) << std::endl;
    return 0;
}
```

Const correctness

```
#include <iostream>

int my_strlen(char *s)
{
    char *p = s;
    while ( ! (*p = 0) ) ++p; // FATAL ERROR
    return p – s;
}

// This program likely cause run-time error
int main()
{
    char t[] = "Hello";
    std::cout << my_strlen(t) << std::endl;
    return 0;
}
```

Const correctness

```
#include <iostream>

int my_strlen(const char *s)
{
    const char *p = s;
    while ( ! (*p = 0) ) ++p; // Compile-time error!
    return p – s;
}


int main()
{
    char t[] = "Hello";
    std::cout << my_strlen(t) << std::endl;
    return 0;
}
```

Const correctness

```
#include <iostream>

int my_strlen(const char *s)
{
    const char *p = s;
    while ( ! (*p == 0) ) ++p; // FIX
    return p – s;
}

// this program is fine
int main()
{
    char t[] = "Hello";
    std::cout << my_strlen(t) << std::endl;
    return 0;
}
```

Side note on defensive programming

```
#include <iostream>

int my_strlen(char *s)
{
    char *p = s;
    while ( ! (0 = *p) ) ++p; // compile-time error!
    return p – s;
}

// This program likely cause run-time error
int main()
{
    char t[] = "Hello";
    std::cout << my_strlen(t) << std::endl;
    return 0;
}
```

Const correctness and pointers

```
      int i = 4;   // not constant, can be modified:
          i = 5;

const int ci = 6;   // const, must be initialized
         ci = 7;   // syntax error, cannot be modified

     int *ip;
         ip = &i;
        *ip = 5;   // ok
```

Const correctness and pointers

```
      int i = 4;    // not constant, can be modified:
          i = 5;

const int ci = 6;   // const, must be initialized
         ci = 7;    // syntax error, cannot be modified

     int *ip;
         ip = &i;
        *ip = 5;    // ok

        if ( input )
          ip = &ci; // ??

        *ip = 7;    // can I do this?
```

Pointer to const

```
      int i = 4;   // not constant, can be modified:
          i = 5;

const int ci = 6;  // const, must be initialized
         ci = 7;   // syntax error, cannot be modified



const int *cip = &ci; // ok
          *cip = 7;   // syntax error
      int *ip = cip;  // syntax error, C++ keeps const

           cip = ip;  // ok, constness gained
          *cip = 5;   // syntax error,
                      // wherever cip points to
```

Pointer constant

```
       int i = 4;   // not constant, can be modified:
           i = 5;

const int ci = 6;   // const, must be initialized
          ci = 7;   // syntax error, cannot be modified


int * const ipc = &i; // ipc is const, must initialize

           *ipc = 5;  // OK, *ipc points is NOT a const


int * const ipc2 = &ci; // syntax error,
                        // ipc is NOT a pointer to const

const int * const cccp = &ci; // const pointer to const
```

const – pointer cheat sheet

- const keyword left to * means pointer to const
  - Can point to const variable
  - Can be changed
  - Pointed element is handled as constant

```
const int ci = 10;
const int * cip = &ci;
int const * pic = &ci;
```

- const keyword right to * means pointer is constant
  - Must be initialized
  - Can not be changed
  - Can modify pointed element

```
int i;
int * const ipc = &i;
```

User-defined types

```
class Date
{
public:
    Date( int year, int month = 1, int day = 1);
    // …
    int getYear();
    int getMonth();
    int getDay();
    void set(int y, int m, int d);
    // …
private:
    int year;
    int month;
    int day;
};
const Date my_birthday(1963,11,11);
      Date curr_date(2015,7,10);
my_birthday = curr_date;  // compile-time error: const
```

User-defined types

```
class Date
{
public:
    Date( int year, int month = 1, int day = 1);
    // …
    int getYear();
    int getMonth();
    int getDay();
    void set(int y, int m, int d);
    // …
private:
    int year;
    int month;
    int day;
};
const Date my_birthday(1963,11,11);
      Date curr_date(2015,7,10);
my_birthday.set(2015,7,10);  // ???
int x = my_birthday.get();   // ???
```

User-defined types

```
class Date
{
public:
    Date( int year, int month = 1, int day = 1);
    // …
    int getYear() const;
    int getMonth() const;
    int getDay() const;
    void set(int y, int m, int d);
    // …
private:
    int year;
    int month;
    int day;
};
const Date my_birthday(1963,11,11);
      Date curr_date(2015,7,10);
my_birthday.set(2015,7,10);  // compile-time error!
int x = my_birthday.get();   // ok
```

Const members

```
class Msg
{
public:
    Msg(const char *t);
    int getId() const { return id; }
private:
    const int id;
    std::string txt;
};

Msg m1("first"), m2("second");
m1.getId() != m2.getId();

MSg::Msg(const char *t)
{
    txt = t;
    id  = getNewId();  // syntax error, id is const
}

//initialization list works
MSg::Msg(const char *t):id(getNextId()),txt(t)
{
}
```

Mutable members

```
struct Point
{
    void getXY(int& x, int& y) const;
    double xcoord;
    double ycoord;
    mutable int read_cnt;
};

const Point a;
++a.read_cnt;  // ok, Point::read_cnt is mutable
```

Mutexes are usually mutables

```
#include <mutex>
struct Point
{
public:
  void getXY(int& x, int& y) const;
  // …
private:
  double  xcoord;
  double  ycoord;
  mutable std::mutex m;
};
// atomic read of point
void getXY(int& x, int& y) const
{
  std::lock_guard< std::mutex > guard(m); // locking of m
  x = xcoord;
  y = ycoord;
} // unlocking m
```

Static const

```
class X
{
  static const int  c1 = 7;    // ok,
  static       int  i2 = 8;    // error: not const
  const        int  c3 = 9;    // error: not static
  static const int  c4 = f(2); // error: non-const init.
  static const float f = 3.14; // error: not integral
};

const int X::c1;  // do not repeat initializer here...
```

Overloading on const

```
template <typename T, ... >
class std::vector
{
public:
    T&       operator[](size_t i);
    const T& operator[](size_t i) const;
    // ...
};
int main()
{
          std::vector<int>   iv;
    const std::vector<int>  civ;
    // ...
    iv[i] = 42;         // non-const
    int i = iv[5];
    int j = civ[5]      // const
    // ...
}
```

STL is const safe

```
template <typename It, typename T>
It find( It begin, It end, const T& t)
{
    while (begin != end) {
        if ( *begin == t )
            return begin;
        ++begin;
    }
    return end;
}
const char t[] = { 1, 2, 3, 4, 5 };
const char *p = std::find( t, t+sizeof(t), 3)
if ( p != t+sizeof(t) )
{
    std::cout << *p; // ok to read
    // syntax error: *p = 6;
}
const std::vector<int>  v(t, t+sizeof(t));
std::vector<int>::const_iterator i = std::find( v.begin(), v.end(), 3);
if ( v.end() != i )
{
     std::cout << *i; // ok to read
    // syntax error: *i = 6;
}
```

STL is const safe

```
//  C++11

std::vector<int>  v1(4,5);
auto i = std::find( v1.begin(), v1.end(), 3);
// i is vector::iterator

const std::vector<int>  v2(4,5);
auto j = std::find( v2.begin(), v2.end(), 3);
// j is vector::const_iterator

auto k = std::find( v1.cbegin(), v1.cend(), 3);
// k is vector::const_iterator
```

Constexpr objects in C++11/14

- Const objects having value known at translation time.
- translation time = compilation time + linking time
- They may have placed to ROM
- Immediately constructed or assigned
- Must contain only literal values, constexpr variables and functions
- The constructor used must be constexpr constructor

Constexpr functions in C++11/14

- Can produce constexpr values when called with compile-time constants.
- Otherwise can run with non-constexpr parameters
- Must not be virtual
- Return type must be literal type
- Parameters must be literal type
- Since C++14 they can be more than a return statement
  - if / else / switch
  - for / ranged-for / while / do-while

Compiling, linking issues

- Inclusion nightmare and compile time
- How to reduce included files
- PIMPL
- Fast PIMPL
- Template code blow
- Linking C and C++ together
- Order of linking issues
- Static initialization/destruction problem

Header files

- Good for defining interface
- Good for breaking circular dependencies
- Must for templates
- Increase compile time
- Break OO principles

Header files

```
#include <iostream>
#include <ostream>
#include <list>
// none of A, B, C, D, E are templates
// Only A and C have virtual functions
#include "a.h"      // class A
#include "b.h"      // class B
#include "c.h"      // class C
#include "d.h"      // class D
#include "e.h"      // class E

class X : public A, private B
{
public:
        X( const C&);
    B   f(int, char*);
    C   f(int, C);
    C&  g(B);
    E   h(E);
    virtual std::ostream& print(std::ostream&) const;
private:
    std::list<C>    clist_;
    D               d_;
};
```

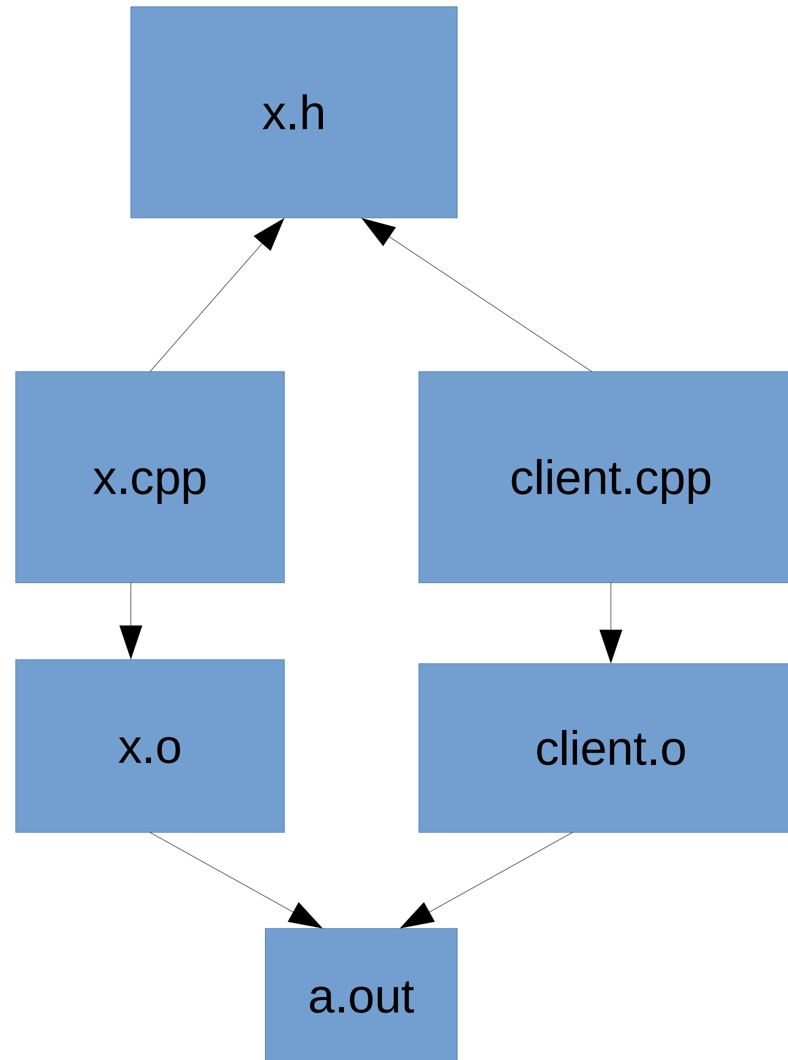
Header files

- Remove <iostream> People automatically include <iostream>, even if input functions never used.
- Replace <ostream> with <iosfwd>. Parameters and return types only need to be forward declared. Because ostream is basic_ostream<char> template, it is not enough to declare.
- Replace "e.h" with forward declaration of class E.
- Leave "a.h" and "b.h": we need a full declaration of the base classes in case of inheritance. The compiler must know the size of bases, whether functions are virtual or not.
- Leave "c.h" and "d.h": list<C> and D are private data members of X.

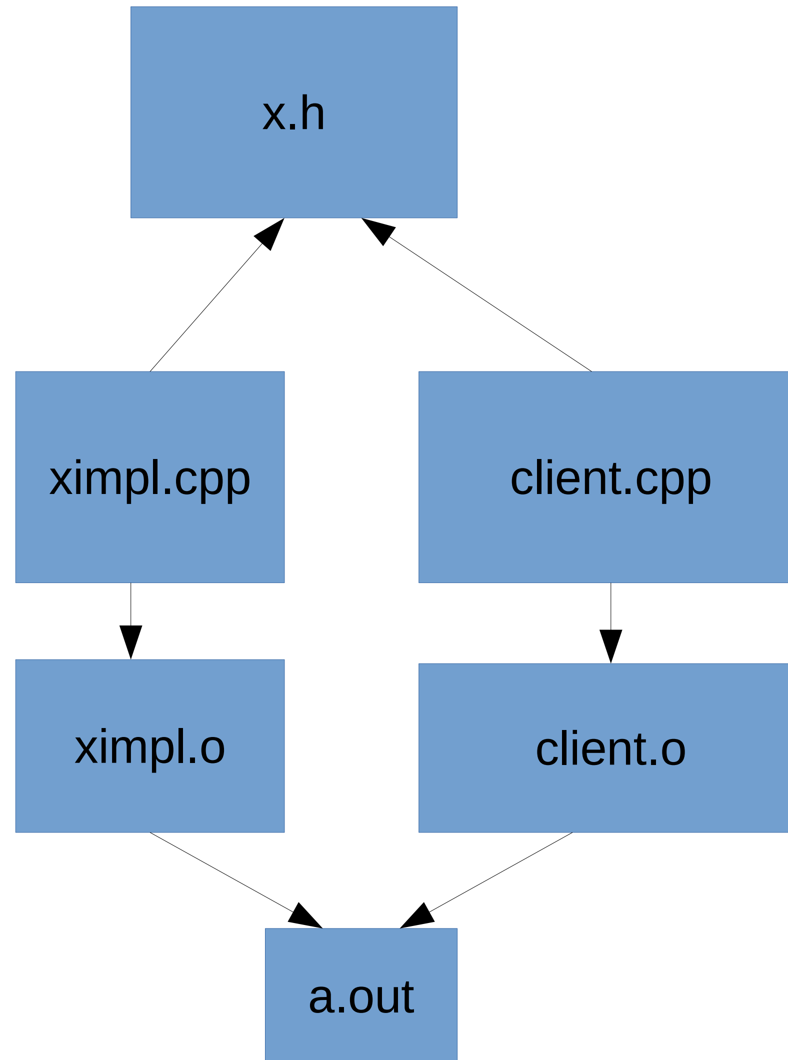
Header files v2

```
#include <iosfwd>
#include <list>
// none of A, B, C, D, E are templates
// Only A and C have virtual functions
#include "a.h"      // class A
#include "b.h"      // class B
#include "c.h"      // class C
#include "d.h"      // class D

class E;
class X : public A, private B
{
public:
        X( const C&);
    B   f(int, char*);
    C   f(int, C);
    C&  g(B);
    E   h(E);
    virtual std::ostream& print(std::ostream&) const;
private:
    std::list<C>    clist_;
    D               d_;
};
```

PIMPL



PIMPL



PIMPL

```
// file x.h
class X
{
    // public and protected members
private:
    // pointer to forward declared class
    struct Ximpl;
    XImpl *pimpl_;  // opaque pointer
};

// file ximpl.cpp
struct XImpl   // not neccessary to declare as "class"
{
    // private members; fully hidden
    // can be changed at without
    // recompiling clients
};
```

PIMPL

```
#include <iosfwd>
#include "a.h"      // class A
#include "b.h"      // class B
class C;
class E;
class X : public A, private B
{
public:
        X( const C&);
    B   f(int, char*);
    C   f(int, C);
    C&  g(B);
    E   h(E);
    virtual std::ostream& print(std::ostream&) const;
private:
    struct Ximpl;
    XImpl *pimpl_; // opaque pointer to forward-declared class
};
// file x.cpp
#include "x.h"
#include "c.h"      // class C
#include "d.h"      // class D
struct Ximpl
{
    std::list<C>    clist_;
    D               d_;
};
```

Removing inheritance

```
#include <iosfwd>
#include "a.h"      // class A
class B;
class C;
class E;
class X : public A  // ,private B
{
public:
        X( const C&);
    B   f(int, char*);
    C   f(int, C);
    C&  g(B);
    E   h(E);
    virtual std::ostream& print(std::ostream&) const;
private:
    struct Ximpl;
    XImpl *pimpl_; // opaque pointer to forward-declared class
};
// file x.cpp
#include "x.h"
#include "b.h"      // class B
#include "c.h"      // class C
#include "d.h"      // class D
struct Ximpl
{
    B               b_;
    std::list<C>    clist_;
    D               d_;
};
```

Fast PIMPL

```
// file x.h
class X
{
  // public and protected members
private:
  static const size_t sizeof_Ximpl = 128;
  char ximpl_[sizeof_XImpl]; // instead opaque pointer
};

// file ximpl.cpp
struct XImpl   // not neccessary to declare as "class"
{
  Ximpl::Ximpl() {
    static_assert (sizeof_XImpl >= sizeof(XImpl));
    new (&ximpl_[0]) Ximpl;
  };
  Ximpl::Ximpl() {
       (reinterpret_cast<XImpl*>(&ximpl_[0]))->~XImpl();
  }
};
```

Template code blow

- Templates are instantiated on request
- Each different template argument type creates new specialization
  - Good: we can specialize for types
  - Bad: Code is generated for all different arguments
  - All template member functions are templates

Template code blow

```
template <class T>
class matrix
{
public:
    int get_cols() const { return cols_; }
    int get_rows() const { return rows_; }
private:
    int cols_;
    int rows_;
    T *elements_;
};

matrix<int>     mi;
matrix<double>  md;
matrix<long>    ml;
```

Template code blow

```
class matrix_base
{
public:
    int get_cols() const { return cols_; }
    int get_rows() const { return rows_; }
protected:
    int cols_;
    int rows_;
};

template <class T>
class matrix : public matrix_base
{
private:
    T *elements_;
};
```

Using C and C++ together

- Object model is (almost) the same
- C++ programs regularly call C libraries
- Issues:
  - Virtual inheritance
  - Virtual functions (pointer to vtbl)
  - Mangled name vs C linkage name

Using C and C++ together

```
//
// C++ header for C/C++ files:
//
#ifdef __cplusplus
extern "C"
{
#endif
    int     f(int);
    double  g(double, int);
    // …
#ifdef __cplusplus
}
#endif
```

Order of linking?

```
// file: a.cpp
static int s = 2;
#include "t.h"
int g()
{
    return t(1);
}
```

```
// file: b.cpp
static int s = 1;
#include "t.h"
int h()
{
    return t(1);
}
```

```
// file: t.h
template <class T>
T t(const T&)
{
    return s;
}
```

```
// file: main.cpp
#include <iostream>
extern int g();
extern int h();
int main()
{
    std::cout << g() <<std::endl;
    std::cout << h() <<std::endl;
    return 0;
}
```

```
$ g++ main.cpp a.cpp b.cpp  &&  ./a.out
2 2
```

Order of linking?

```
// file: a.cpp
static int s = 2;
#include "t.h"
int g()
{
    return t(1);
}
```

```
// file: b.cpp
static int s = 1;
#include "t.h"
int h()
{
    return t(1);
}
```

```
// file: t.h
template <class T>
T t(const T&)
{
    return s;
}
```

```
// file: main.cpp
#include <iostream>
extern int g();
extern int h();
int main()
{
    std::cout << g() <<std::endl;
    std::cout << h() <<std::endl;
    return 0;
}
```

```
$ g++ main.cpp b.cpp a.cpp  &&  ./a.out
1 1
```

Static initialization/destruction

- Static objects inside translation unit constructed in a well-defined order
- No ordering between translation units
- Issues:
  - (1) Constructor of static refers other source's static
  - Destruction order
- Lazy singleton solves (1)
- Phoenix pattern solves (2)

Constructor/Destructor

- In OO languages constructors to initialize objects
- Other style: using factory objects
- Constructors are also
  - Conversion operators
  - Create temporaries
- Destructors are to free resources
  - Resource: much more than memory
  - e.g. unlock, fclose, ...

Constructor/Destructor

- Constructor as conversion operator
- How to define operators
- Initialization list
- Destructors / array destruction
- Cloning

A simple date class

```
#inlcude “date.h”

int main()
{
    date today();   // current date from OS
    date exam1( 2016, 1, 15);  // a date
    date new_year( 2016);      // date(2016,1,1)

    std::string s(“2016.2.1”);
    date feb1(s.c_str());      // date(2016,2,1)


    if ( exam1 < date(2016,2) )
      …

    date jan1 = “2016”; // syntax error!
};
```

A simple date class

```
class date
{
public:
    // default constructor
    date();
    // constructor
    date( int y, int m=1, int d=1);
    // explicite constructor
    explicit date( const char *s);
    // …
private:
    // …
    int year;
    int month;
    int day;
};
```

Where to define operators

- Theory says: bind with class
  - Member operators
- Some operators can't be members
  - `std::ostream& operator<<(std::ostream&,const X&)`
- Sometimes members creates unwanted dependences

```
std::getline(std::basic_istream&,std::basic_string&)
```

- Sometime operators should be symmetric

Symmetry

```
#inlcude “date.h”

int main()
{
    date today();    // current date from OS


    if ( today < date(2016) )  // works

    if ( today < 2016 )        // works

    if ( 2016 < today )        // not working
                               // if operator< is member
};
```

Operators

```
// date.h
class date
{
public:
    // …
private:
    // …
    int year;
    int month;
    int day;
};

       bool operator<( date d1, date d2);
inline bool operator==(date d1,date d2)   { ... }
inline bool operator!=( date d1, date d2) { ... }
inline bool operator<=( date d1, date d2) { ... }
inline bool operator>=( date d1, date d2) { ... }
inline bool operator>( date d1, date d2)  { ... }
```

Creation of subobjects

```
// date.h
class date
{
public:
    date( int y, int m, int d);
private:
    // …
    int year;
    int month;
    int day;
};

date::date(int y, int m, int d) : day(d),month(m),year(y)
{ . . . }

// rearranged to declaration order:
date::date(int y, int m, int d) : year(y),month(m),day(d)
{ . . . }
```

Destructors

- Should be virtual?
  - When used in polymorphic way
  - When there is multiply inheritance

```
class Base
{
public:
    virtual ~Base();
};
class Der : public Base
{
public:
    virtual ~Der();
};
```

```
void f()
{
    Base *bp = new Der;
    // …
    delete bp;
}
```

Arrays are not polymorphic!

```
struct Base
{
    Base() { cout << "Base" << " "; }
    virtual ~Base() { cout << "~Base" << endl; }
    int i;
};
struct Der : public Base
{
    Der() { cout << "Der" << endl; }
    virtual ~Der() { cout << "~Der" << " "; }
    int it[10]; // sizeof(Base) != sizeof(Der)
};
int main()
{
    Base *bp = new Der;
    Base *bq = new Der[5];

    delete    bp;
    delete [] bq;   // this causes runtime error
}
```

Cloning – “Virtual” constructors

- Constructors are not virtual
- But sometimes we need similar behavior

```
std::vector<Base*> source;
std::vector<Base*> target;

source.push_back(new Derived1());
source.push_back(new Derived2());
source.push_back(new Derived3());

// should create new instances of the
// corresponding Derived classes and
// place them to target
deep_copy( target, source);
```

Wrong approach

```
deep_copy( std::vector<Base*> &target,
           const std::vector<Base*> &source)
{
  for(auto i = source.beg(); i!=source.end(); ++i)
  {
    target.push_back(new Base(**i)); // creates Base()
  }
}
```

Cloning

```
class Base
{
public:
  virtual Base* clone() { return new Base(*this); }
};
class Derived : public Base
{
public:
  virtual Derived* clone() { return new Derived(*this); }
};

deep_copy( std::vector<Base*> &target,
           const std::vector<Base*> &source)
{
  for(auto i = source.beg(); i!=source.end(); ++i)
  {
    target.push_back(i->clone()); // inserts Derived()
  }
}
```

Pointers and references

- Pointer arithmetics
- Pointers and arrays
- References
  - Parameter passing with reference
  - Functions returning with reference
- Move semantics in C++11

Arrays

- An array is strictly continuous memory area
- Arrays do not know their size, but: sizeof(t) / sizeof(t[0])
- Array names are converted to pointer value to first element
- No multidimensional arrays. But are array of arrays.
- No operations on arrays, only on array elements

```
int t[2][4];

assert(sizeof(t) == 8*sizeof(int));
assert(sizeof(t[0] == 4*sizeof(int));
assert(sizeof(t[0][0])==sizeof(int));

t[0][1] = t[1][1];
// t[0] = t[1];  syntax error
```

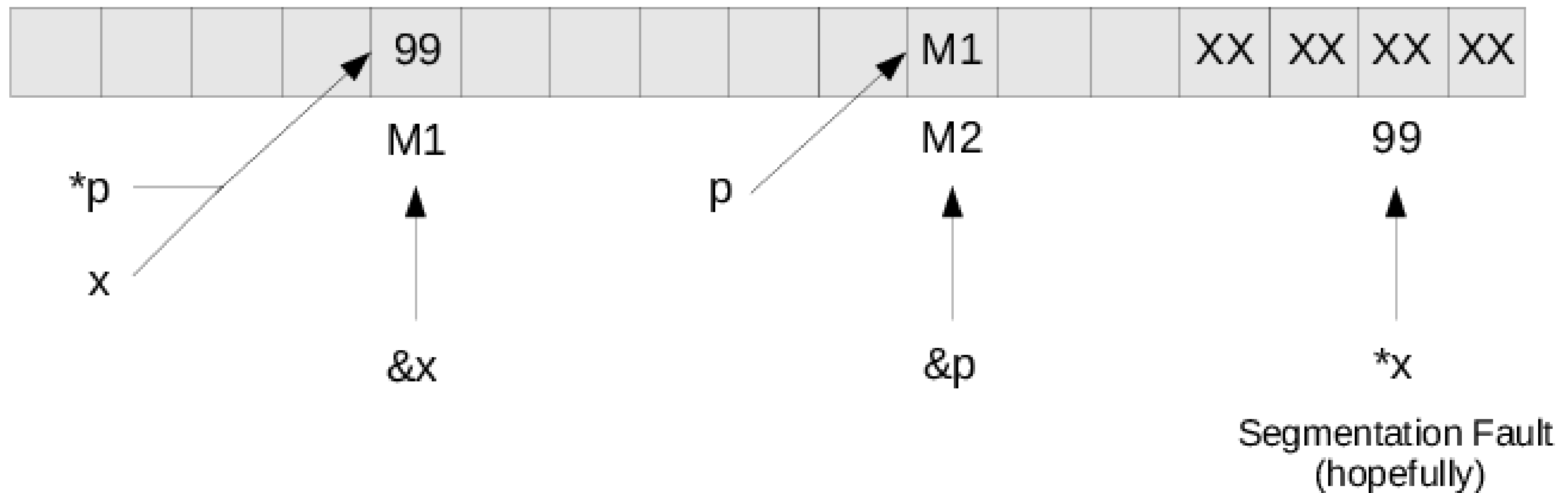
Pointers

Pointers in C

C Code

```
int x;
int *p;

x = 99;    //holds a value
p = &x;    //holds an address of a value
```

Memory



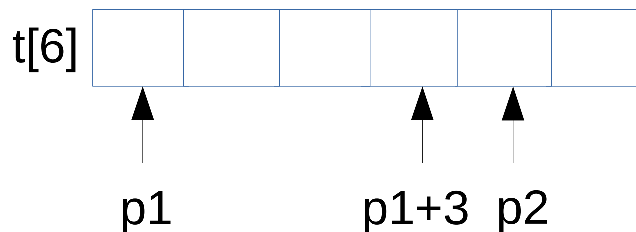
Pointers

- Pointer is a value that refers to a(n abstract) memory location
- Pointers can refer to ANY valid memory locations (unlike e.g. PASCAL)
- NULL (declared in <stdio.h>) is a universal null-pointer value
- Non-null pointers are taken as TRUE value

```
char *ptr = (char *) malloc(1024);

if ( ptr )
{
   // ptr != NULL here
}
```

Pointer arithmetics

- Integers can be added and subtracted from pointers
- Pointers pointing to the same array can be subtracted
- Pointer arithmetics depends on the pointed type!
- No pinter arithmetics on void *




```
int t[6];
int *p1 = &t[0];
int *p2 = &t[4];

assert( &t[3] == p1+3 );
assert( p2 – p1 == 4 );
assert( p1 + 4 == p2 );
```

```
  p = &t[0];
  p = t;
  p + k == &t[k]
*(p + k) == t[k]
```

Pointers and arrays

- Pointers and array names can be used similarly in expressions

```
int t[10];
int i = 3;

int *p = t;       // t is used as pointer to the first element

p[i] = *(t + i); // p used as array, t is used as pointer
```

But pointers ARE NOT EQUIVALENT TO arrays !!!

Pointers and arrays

```
// s1.c
#include <stdio.h>

int t[] = {1,2,3};

void f( int *par)
{
  printf(“%d”,par[1]);
  printf(“%d”,t[1]);
}
int main()
{
  int *p = t;
  printf("%d",t[1]);
  printf("%d",p[1]);
  f(t);
  g(t);
}
```

```
// s2.c
#include <stdio.h>

extern int *t;

void g( int *par)
{
  printf(“%d”,par[1]);

  // the program crashes here
  printf(“%d”,t[1]);
}
```

Nullptr

- Nullptr is a new literal since C++11 of type std::nullptr_t
- Helps to overload between pointers and integer
- Automatic conversions from any type null pointers and from NULL

```
void f(nullptr); // 1
void f(int);     // 2

f(0);          // calls 2
f(nullptr);    // calls 1
```

Reference

- In modern languages definitions hide two important but orthogonal concepts:
  - Allocate memory for a variable
  - Bind a name with special scope it
- In most languages this is unseparable
- In C++ we can do separate the two steps

```
void f()
{
  int i;                  // allocate memory, bind i as name
  int &i_ref = i;         // binds a new name
  int *iptr = new int;    // allocate memory, no binded name
  int &k = *iptr;         // binds a new name to unnamed int area
  delete iptr;            // memory invalidated name k still lives
  k = 5;                  // compiles, later run-time error
}                         // k goes out of scope
```

Pointer vs reference

- Pointer has extremal element: nullptr
  - nullptr means: pointing to no valid memory
- Reference always should refer to valid memory
  - Use exception, if something fails

```
if ( Derived *dp = dynamic_cast<Derived*>(bp) )
{
    // ...
}
if ( Derived &dp = dynamic_cast<Derived&>(bp) ) // may throw
{
    // ...
}
catch(bad_cast &be) { . . . }
```

Parameter passing

- Parameter passing in C++ follows initialization semantics
  - Value initialization copies the object
  - Reference initialization just set up an alias

```
void f1( int x, int y) { ... }
void f2( int &xr, int &yr) { ... }

int i = 5, j = 6;

f1( i, j);    int x = i;  // creates local x and copies i to x
              int y = j;  // creates local y and copies j to y

f2( i, j);    int &xr = i; // binds xr name to i outside
              int &yr = j; // binds yr name to j outside
```

Swap before move semantics

```
void swap( int &x, int &y)
{
  int tmp = x;
  x = y;
  y = tmp;
}

int main()
{
  int i = 5;
  int j = 6;

  swap( i, j);
  assert(i==6 && j==5);
}
```

Reference binding

- Non-const (left) reference binds only to left value
- Const reference binds to right values too

```
int i = 5, j = 6;

swap(i,j);      // ok
swap(i,7);      // error: could not bind reference to 7
int &ir1 = 7;   // error: could not bind reference to 7

swap(i,3.14);          // error: int(3.14) creates non-left value
int &ir2 = int(3.14); // error: int(3.14) creates non-left value

const int &ir3 = 7;          // ok
const int &ir4 = int(3.14); // ok
```

Returning with reference

- By default C++ functions return with copy
- Returning reference just binds the function result to an object

```
int f1()
{
  int i = 5;
  return i;   // ok, copies i before evaporating
}

int &f2()
{
  int i = 5;
  return i;   // oops, binds to evaporating i
}
```

Usage example

```
class date
{
public:
  date& date::setYear(int y)  { _year = y;  return *this; }
  date& date::setMonth(int m) { _month = m; return *this; }
  date& date::setDay(int d)   { _day = d;   return *this; }

  date& date::operator++() { ++_day; return *this; }
  Date date::operator++(int) // should return temporary
                         { date c(*this); ++_day; return *c; }
private:
  int _year;
  int _month;
  int _day;
};

void f()
{
  date d;
  ++d.setYear(2011).setMonth(11).setDay(11); // still reference
}
```

Usage example

```
template <typename T>
class matrix
{
public:
       T& operator()(int i, int j)      { return v[i*cols+j]; }
 const T& operator()(int i,int j) const { return v[i*cols+j]; }
    matrix& operator+=( const matrix& other)
    {
        for (int i = 0; i < cols*rows; ++i)
            v[i] += other.v[i];
    }
private:
    // ...
    T* v;
};
template <typename T> matrix<T>  // returns value
operator+(const matrix<T>& left, const matrix<T>& right)
{
    matrix<T> result(left);  // copy constructor
    result += right;
    return result;
}
```

Advanced memory handling

- Storage classes in C++
- The new and delete operators
- Overloading new and delete
- New and delete expressions
- Objects with restricted storage classes
- The RAII idiom

Storage classes in C++

- String literals are readonly objects
- Automatic (local) variables
- Global (namespace) static variables
- Local static variables
- Dynamic memory with new/delete or malloc/free
- Temporaries
- Arrays
- Subobjects (non-static class members)

Temporaries

- Created when evaluating an expression
- Guaranteed to live until the full expression is evaluated

```
void f( string &s1, string &s2, string &s3)
{
  const char *cs = (s1+s2).c_str();
  cout << cs;     // Bad!!

   if ( strlen(cs = (s2+s3).c_str()) < 8 && cs[0] == 'a' ) // Ok
        cout << cs;     // Bad!!
}
void f( string &s1, string &s2, string &s3)
{
    cout << s1 + s2;
    const string &s = s2 + s3; // binding to name keeps temporary
                               // alive until name goes out of scope
    if ( s.length() < 8 && s[0] == 'a' )
        cout << s;  // Ok
}
// s1+s2 destroyes here: when the const ref goes out of scope
```

New and delete

- New and delete expressions
- New and delete operators

```
void f( string &s1, string &s2, string &s3)
{
  try
  {
    int *ip = new int(42);  // new expression
    int *ap = new int[10];

    int *ptr = static_cast<int *>(::operator new(sizeof(int)));
  }
  catch(std::bad_alloc e) { ... }

  ::operator delete(ptr);

  delete ip;
  delete [] ap;
}
```

New and delete operators

- May throw std::bad_alloc exception
- Returns void*

```
namespace std
{
  class bad_alloc : public exception { /* ... */ };
}

void* operator new(size_t);     // new() may throw bad_alloc
void  operator delete(void *)  // delete() never throws
```

Nothrow version

- Returns nullptr if allocation is unsuccessful

```
// indicator for allocation that doesn't throw exceptions
struct nothrow_t {};
extern const nothrow_t nothrow;

// what to do, when error occurs on allocation
typedef void (*new_handler)();
new_handler set_new_handler(new_handler new_p) throw();

// nothrow version
void* operator new(size_t, const nothrow_t&);
void  operator delete(void*, const nothrow_t&);

void* operator new[](size_t, const nothrow_t&);
void  operator delete[](void*, const nothrow_t&);
```

Placement new

- Never allocate / deallocate memory

```
// placement new and delete
void* operator new(size_t, void* p)   { return p; }
void  operator delete(void* p, void*) { }

void* operator new[](size_t, void* p)   { return p; }
void  operator delete[](void* p, void*) { }

#include <new>
void f()
{
  char *cp = new char[sizeof(C)];
  for( long long i = 0; i < 10000000; ++i)
  {
     C *dp = new(cp) C(i);
     // ...
     dp->~C();
  }
  delete [] cp;
  return 0;
}
```

Overloading new and delete

- New and delete operators can be overloaded at two level
  - Class level (automatically static member functions)
  - Namespace level

```
struct node
{
          node( int v)  { val = v; left = rigth = 0; }
    void  print() const { cout << val << " "; }

    /* static */ void *operator new( size_t sz) throw (bad_alloc);
    /* static */ void  operator delete(void *p) throw();

    int   val;
    node *left;
    node *right;
};
```

Overloading new and delete

```
// member new and delete as static member
void *node::operator new( size_t sz) throw (bad_alloc)
{
    return ::operator new(sz);
}
void node::operator delete(void *p) throw()
{
    ::operator delete(p);
}
// global new and delete
void *operator new( size_t sz) throw (bad_alloc)
{
    return malloc(sz);
}
void operator delete( void *p) throw ()
{
    free(p);
}
```

Extra parameters for new

- One can define new and delete with extra parameters
  - Only new expression can pass extra parameters

```
struct node
{
          node( int v);
    void  print() const;

    static void *operator new( size_t sz, int i);
    static void  operator delete(void *p, int i);

    int   val;
    node *left;
    node *right;
};
void f()
{
    int i = 3;
    node *r = new(i+3) node(i);
}
```

New expression

New expression do the following 3 steps when called as new X

- Allocate memory for X (usually calling operator new(sizeof(X)))
- Calls the constructor of X passing parameters if exists
- Converts pointer to the new object from void* to X* and returns

- But, what if
  - Operator new throws bad_alloc?
  - The constructor throws anything

```
X *ptr;
try
{
    ptr = new X(par1, par2);
}
catch( ... )
{
    // handle exceptions. Memory leak when constructor throws???
}
```

Objects only in heap

- Sometimes restriction for storage location is useful

```
// Class should be allocated only in heap
class X
{
public:
    X() {}
    void destroy() const { delete this; }
protected:
    ~X() {}
};
class Y : public X { };
//  class Z { X xx; };  // use pointer!
void f()
{
    X* xp = new X;
    Y* yp = new Y;

    delete xp;      // syntax error
    xp->destroy();  // ok
}
```

Objects never in heap

- Sometimes restriction for storage location is useful

```
// Class should be allocated only not in heap
class X
{
private:
    static void *operator new( size_t);
    static void operator delete(void *);
    // Use: static void operator delete(void *) = delete; in C++11
};

class Y : public X { };
//  class Z { X xx; };  // use pointer!
void f()
{
    X* xp = new X;  // error
    X  x;           // ok
}
```

RAII

- Resource Acquisition Is Initialization
- The idea: keep a resource is expressed by object lifetime

```
// is this correct?
void f()
{
    char *cp = new char[1024];

    g(cp);
    h(cp);

    delete [] cp;
}
```

RAII

- Resource Acquisition Is Initialization
- The idea: keep a resource is expressed by object lifetime

```
// is this maintainable?
void f()
{
    char *cp = new char[1024];

    try
    {
      g(cp);
      h(cp);
      delete [] cp;
    }
    catch (...)
    {
      delete [] cp;
    }
}
```

RAII

- Constructor allocates resource
- Destructor deallocates

```
// RAII
struct Res
{
    Res(int n) { cp = new char[n]; }
    ~Res() { delete [] cp; }
    char *getcp() const { return cp; }
};

void f()
{
    Res res(1024);

    g(res.getcp());
    h(res.getcp());
}
// resources will be free here
```

RAII

- Should be careful when implementing RAII
- Destructor calls only when living object goes out of scope
- Object lives only when constructor has successfully finished

```
// But be care:
struct BadRes
{
    Res(int n) { cp = new char[n]; ... init(); ... }
    ~Res()     { delete [] cp; }
    char *cp;
    void init()
    {
       ...throw XXX;
    }
};
```

Typical RAII solutions

- Smart pointers for memory handling
- Guards for locking
- ifstream, ofstream objects for file-i/o
- std::containers

```
class X
{
public:
    void *non_thread_safe();
private:
    Mutex lock_;
};

void *X::non_thread_safe();
{
    Guard<Mutex> guard(lock_);
    /* critical section */
}
```

Smart pointers

- The deprecated auto pointer
- How smart pointers work?
- Unique_ptr
- Shared_ptr and weak_ptr
- Make_ functions
- Shared pointer from this
- Traps and pitfalls

Auto_ptr

- The only smart pointer in C++98/03
- Cheap, ownership-based
- Not works well with STL containers and algorithms
- Not works with arrays
- Deprecated in C++11 Don't use it!

```
int main()
{
    auto_ptr<int> p(new int(42));
    auto_ptr<int> q;
    // p: 42   q: NULL

    q = p;
    // p: NULL q: 42

    *q += 13;   // change value of the object q owns
    p = q;
    // p: 55   q: NULL
}
```

How inheritance works?

- Raw pointers: assign Derived* to Base* works
- But auto_ptr<Derived> is not inherited from auto_ptr<Base> !
- We need a bit of trick here

```
template<class T>
class auto_ptr
{
private:
  T* ap;    // refers to the actual owned object (if any)
public:
  typedef T element_type;

  explicit auto_ptr (T* ptr = 0) : ap(ptr) { }
  auto_ptr (auto_ptr& rhs) : ap(rhs.release()) { }
  template<class Y> auto_ptr(auto_ptr<Y>& rhs) : ap(rhs.release()){ }

  auto_ptr& operator=(auto_ptr& rhs) { ... }
  template<class Y> auto_ptr& operator= (auto_ptr<Y>& rhs) { ... }
};
```

Unique_ptr

- Single ownership pointer (similar to auto_ptr)
- But carefully designed to work with STL and other language features
- Moveable but not copiable
- Deleter type parameter

```
#include <memory>
template< class T, class Deleter = std::default_delete<T>>
class unique_ptr;

template <class T, class Deleter>
class unique_ptr<T[],Deleter>;

void f()
{
  std::unique_ptr<MyClass>   up1(new MyClass());  // * and ->
  std::unique_ptr<MyClass[]> up2(new MyClass[n]); // []
  ...
}  // proper delete called here
```

Unique_ptr

```
#include <memory>

void f()
{
    std::unique_ptr<Foo> up(new Foo());  // up is the only owner
    std::unique_ptr<Foo> up2(up);    // syntax error: can't copy
unique_ptr

    std::unique_ptr<Foo> up3;        // nullptr: not owner

    up3 = up;    // syntax error: can't assign unique_ptr
    up3 = std::move(up);        // ownership moved to up3

} // up3 destroyed: Foo object is destructed
  // up  destroyed: nop
```

Abstract factory pattern

```
#include <memory>

class Base { ... };
class Derived1 : public Base { ... };
class Derived2 : public Base { ... };


template <typename... Ts>
std::unique_ptr<Base>  makeBase( Ts&&... params)  { ... }

void f() // client code:
{
  auto pBase = makeBase( /* arguments */ );
}
// destroy object
```

Abstract Factory Pattern

```
auto delBase = [](Base *pBase)
               {
                   makeLogEntry(pBase);
                   delete pBase; // delete object
               };

template <typename... Ts>
std::unique_ptr<Base, decltype(delBase)>  makeBase( Ts&&... params)
{
  std::unique_ptr<Base, decltype(delBase)> pBase(nullptr, delBase);

  if ( /* Derived1 */ )
  {
    pBase.reset(new Derived1( std::forward<Ts>(params)...) );
  }
  else if ( /* Derived2 */ )
  {
    pBase.reset(new Derived2( std::forward<Ts>(params)...) );
  }
  return pBase;
}
```

Evaluation

- The sizeof(unique_ptr<>) without deleter is == sizeof(raw pointer)
- If there is deleter with state, the size increases
- If no state, then no size penalty (e.g. lambda with no capture).
- If no state, but function pointer is used as deleter: sizeof(funptr) added
- Prefer unique_ptr when possible
- Can be used in standard containers when polymorphic use needed
- Cheap
- No downcast operation :(

shared_ptr

- Shared ownership pointer with reference counter
- Copy constructible and assignable
- No array specializations (e.g. no shared_ptr<T[ ]>)
- Deleter type parameter

```
#include <memory>
template< class T, class Deleter = std::default_delete<T>>
class shared_ptr;

void f()
{
  std::shared_ptr<MyClass>  sp1(new MyClass());  // * and ->
  std::shared_ptr<MyClass>  sp2(new MyClass[n],  // * and ->
                                    std::default_delete<test[]>());
  ...
}  // proper delete called here
```

shared_ptr

```
void f()
{
  std::shared_ptr<int> p1(new int(5));
  std::shared_ptr<int> p2 = p1; // now both own the memory.

  p1.reset(); // memory still exists, due to p2.
  p2.reset(); // delete the memory, since no one else owns.
}
```

```
T & operator*() const;  // never throws
T * operator->() const; // never throws
T * get() const;        // never throws

bool unique() const;    // never throws
long use_count() const; // never throws
```

weak_ptr

- Not owns the memory
- But part of the “sharing group”
- No direct operation to access the memory
- Can be converted to shared_ptr with lock()

```
long use_count() const;
bool expired() const;     // use_count() == 0

shared_ptr<T> lock() const;
// return expired() ? shared_ptr<T>() : shared_ptr<T>(*this)

void reset();
```

Using lock()

```
void f()
{
    std::shared_ptr<X> ptr1 = std::make_shared<X>();
    std::shared_ptr<X> ptr2 = ptr1;

    std::weak_ptr<X>   wptr = ptr2;

    if ( auto sp = wptr1.lock() )
       // use sp
    else
       // expired
} // destructors are called here
```

Using lock()

```
int main ()
{
  std::shared_ptr<int> sp1,sp2;
  std::weak_ptr<int>   wp;
                                       // sharing group:
                                       // --------------
  sp1 = std::make_shared<int> (20);    // sp1
  wp = sp1;                            // sp1, wp

  sp2 = wp.lock();                     // sp1, wp, sp2
  sp1.reset();                         //      wp, sp2

  sp1 = wp.lock();                     // sp1, wp, sp2
}
```

Make functions

```
// For unique_ptr
// default constructor of T
std::unique_ptr<T> v1 = std::make_unique<T>();
// constructor with params
std::unique_ptr<T> v2 = std::make_unique<T>(x,y,z);
// array of 5 elements
std::unique_ptr<T[]> v3 = std::make_unique<T[]>(5);

// For shared_ptr
void f()
{
    std::unique_ptr<Foo> up(new Foo());  // up is the only owner

    if ( up )  // owner or not
    {
        *up = ...;   // use the object
    }
}
```

Enable shared from this

```
#include <memory>
#include <cassert>

class Y: public std::enable_shared_from_this<Y>
{
public:

    std::shared_ptr<Y> f()
    {
        return shared_from_this();
    }
};

int main()
{
    std::shared_ptr<Y> p(new Y);
    std::shared_ptr<Y> q = p->f();
    assert(p == q);
    assert(!(p < q || q < p)); // p and q must share ownership
}
```

Trap: exception safety

```
int f(); // may throw exception

// possible memory leak
std::pair<std::unique_ptr<MyClass>,int> foo()
{
  return std::make_pair(std::unique<MyClass>(new MyClass()), f());
}
```

Trap: exception safety

```
int f(); // may throw exception

// possible memory leak
std::pair<std::unique_ptr<MyClass>,int> foo()
{
  return std::make_pair(std::unique<MyClass>(new MyClass()), f());
}
```

```
1. Runs new MyClass
2. Runs f() and throw exception
3. std::unique_ptr constructor is not called
```

Trap: exception safety

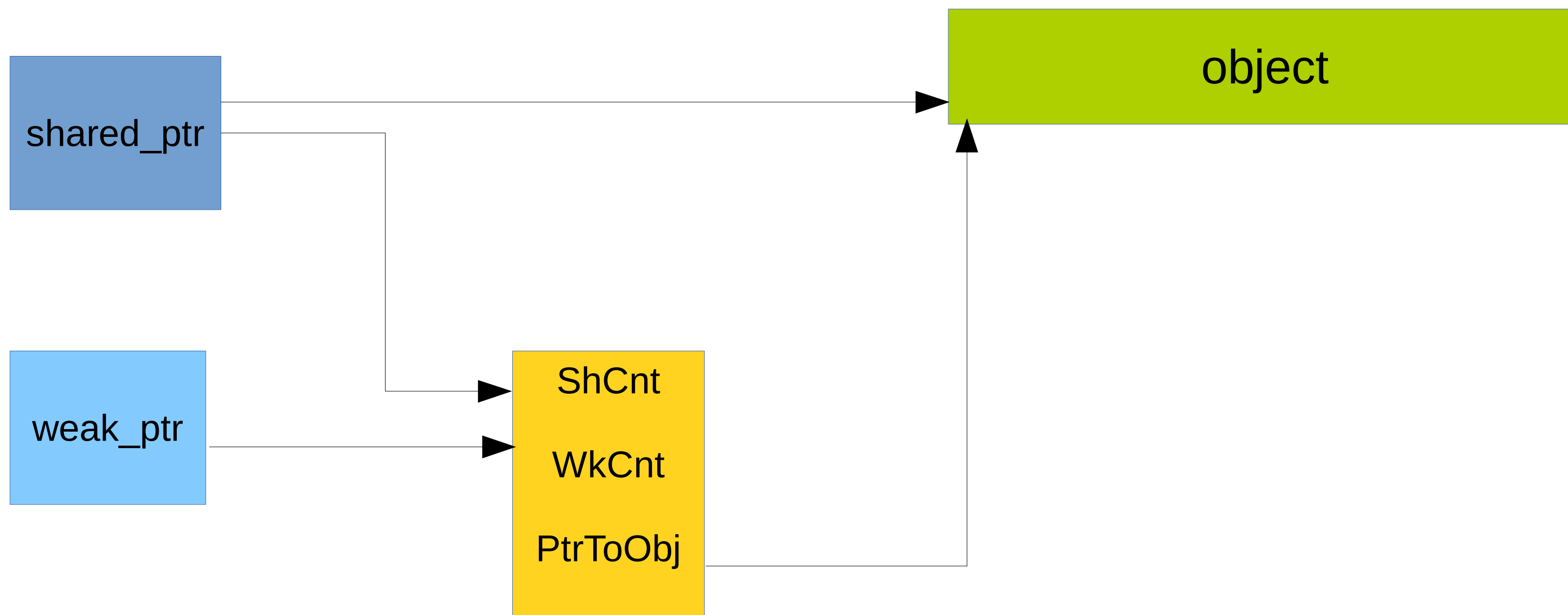
```
int f(); // may throw exception

// possible memory leak
std::pair<std::unique_ptr<MyClass>,int> foo()
{
  return std::make_pair(std::unique<MyClass>(new MyClass()), f());
}


int f(); // may throw exception

// safe
std::pair<std::unique_ptr<MyClass>,int> foo()
{
  return
  std::make_pair(std::make_unique<MyClass>(new MyClass()), f());
}
```

Trap: exception safety

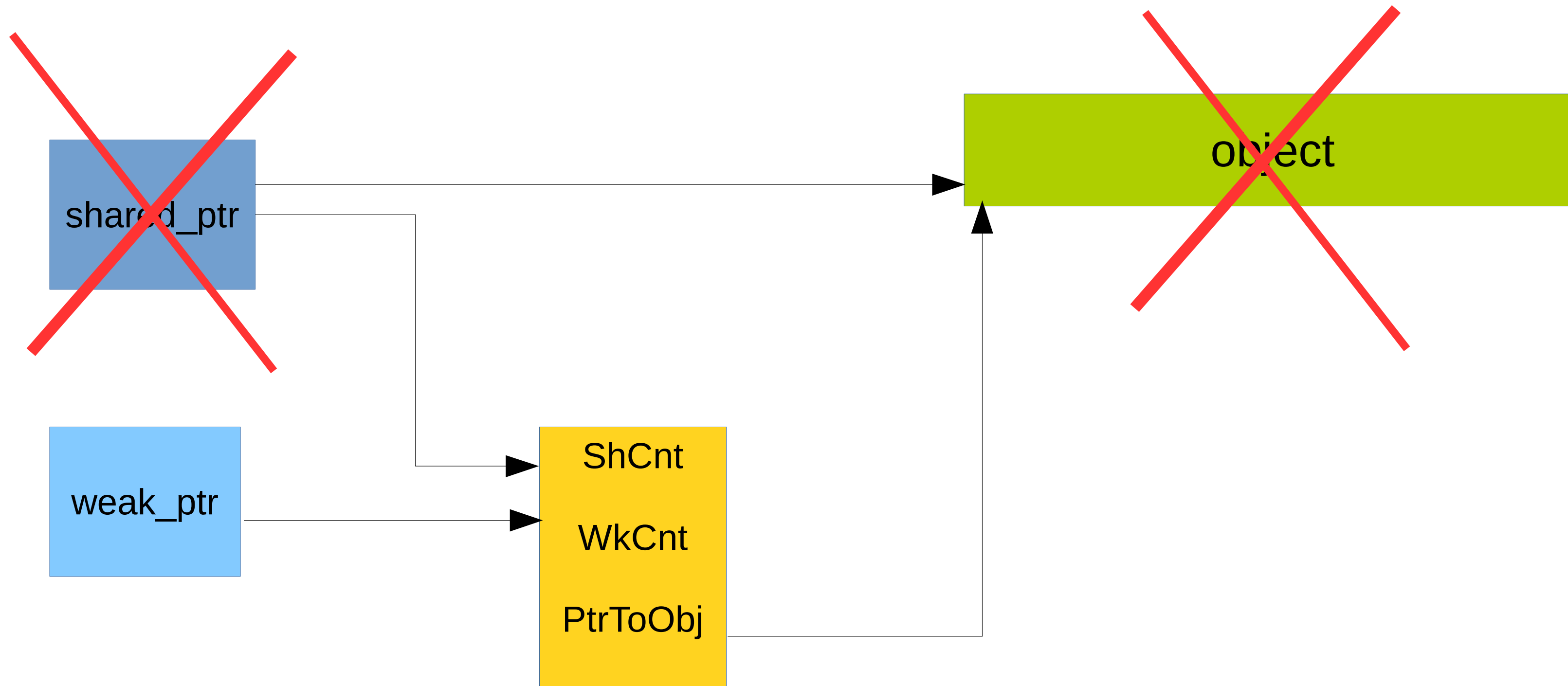
```
int f(); // may throw exception

// possible memory leak
std::pair<std::unique_ptr<MyClass>,int> foo()
{
  return std::make_pair(std::unique<MyClass>(new MyClass()), f());
}


int f(); // may throw exception

// safe
std::pair<std::unique_ptr<MyClass>,int> foo()
{
  return
  std::make_pair(std::make_unique<MyClass>(new MyClass()), f());
}
```

No news – good news!

Trap: overuse of memory

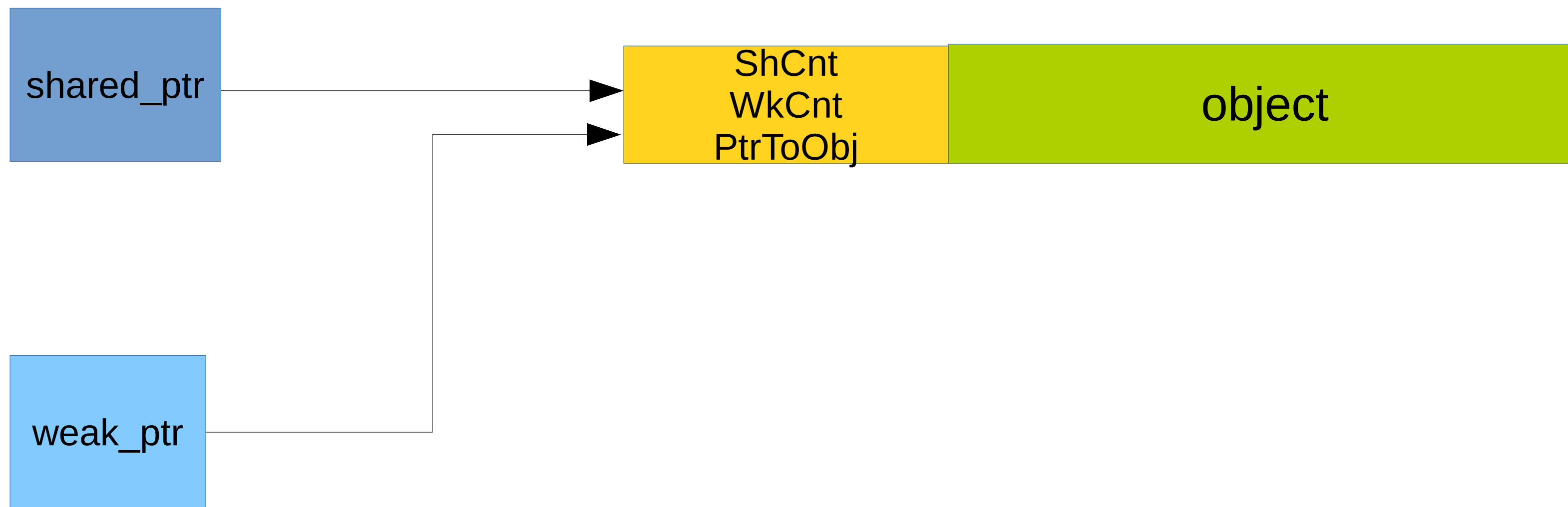


Trap: overuse of memory

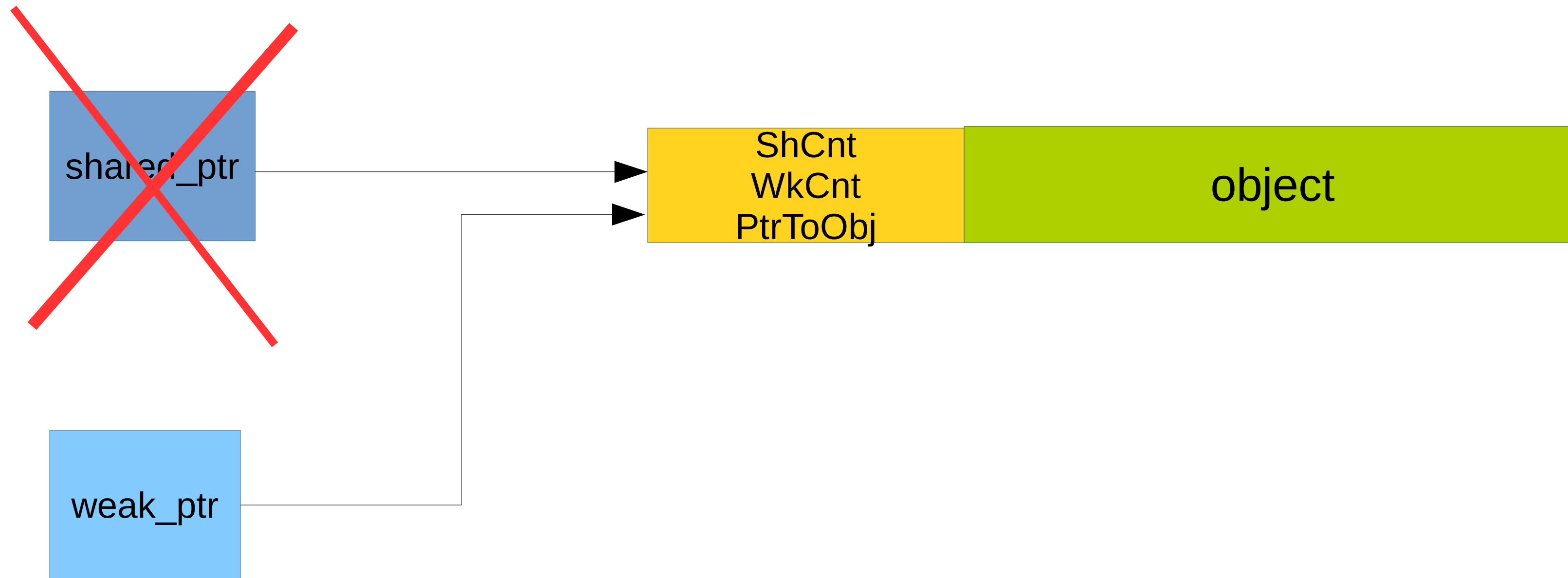


Trap: overuse of memory

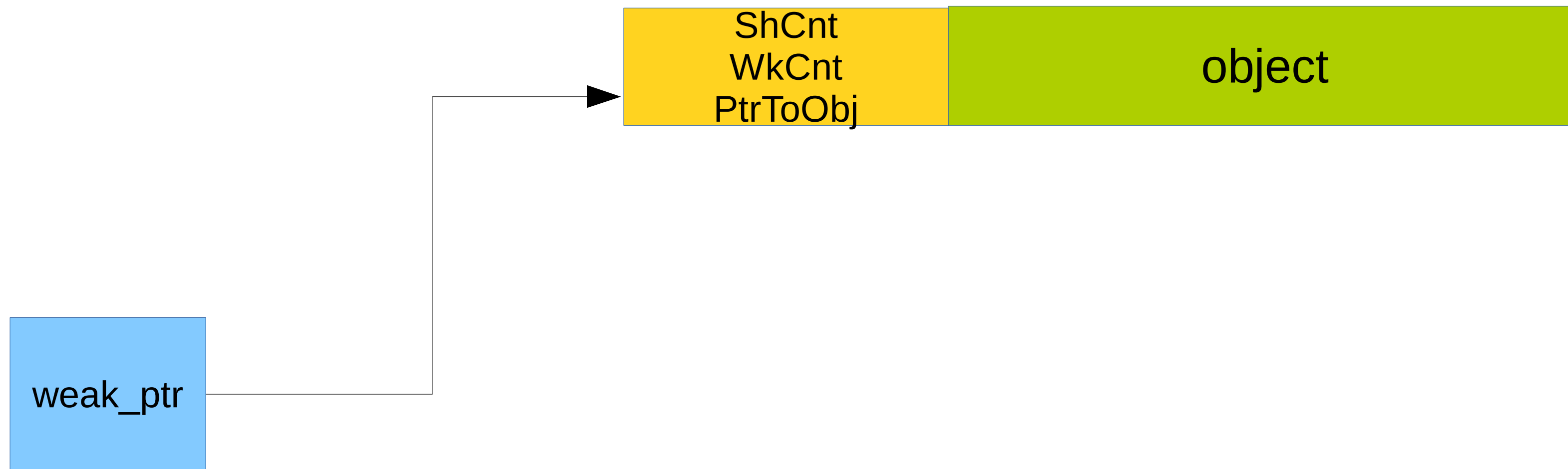
Trap: overuse of memory



Trap: overuse of memory



Trap: overuse of memory



When NOT to use make_*

- Both
  - You need custom deleter
  - You want to use braced initializer
- std::unique_ptr
  - You want custom allocator
- std::shared_ptr
  - Long living weak_ptrs
  - Class-specific new and delete
  - Potential false sharing of the object and the reference counter

Templates

- From macros to templates
- Parameter deduction, instantiation,specialization
- Class templates, partial specialization
- Explicit instantiation
- Dependent types
- Scope resolution, lookup
- Mixins
- CRTP (Curiously Recurring Template Pattern)
- Variadic templates in C++11

Templates

- Originally Stroustrup planned only Macros
- Side effects are issue with Macros: no types known
- Templates are integrated to C++ type system
- Templates are not functions, they are skeletons
- Parameter deduction
- Instantiation

Templates

```
template <typename T>
void swap( T& x, T& y)
{
    T temp = x;
    x = y;
    y = temp;
}
template <typename T>
T max( T a, T b)
{
    if ( a > b )
        return a;
    else
        return b;
}
void f()
{
    int i = 3, j = 4, k;
    double f = 3.14; g = 5.55, h;
    k = max(i,j);
    h = max(f,g);
    h = max(i,f);
}
```

Templates with more types

```
template <class T, class S>
T max( T a, S b)
{
    if ( a > b )
        return a;
    else
        return b;
}

int f()
{
  int     i = 3;
  double  x = 3.14;

  z = max( i, x);
}
```

No deduction on return type

```
template <class R, class T, class S>
R max( T a, S b, R)
{
    if ( a > b )
        return a;
    else
        return b;
}
z = max( i, x, 0.0);

template <class R, class T, class S>
R max( T a, S b)
{
    if ( a > b )
        return a;
    else
        return b;
}
z = max<double>( i, x);               // ok, returns 3.14
k = max<long, long, int>( i, x);      // converts to long and int
k = max<int, int, int>( i, j);
```

Template overloading

```
template <class T>  T   max(T,T);
template <class R, class T, class S>    R   max(T,S);

template <>
const char *max( const char *s1, const char *s2)
{
    return  strcmp( s1, s2) < 0 ? s1 : s2;
}

int i = 3, j = 4, k;
double x = 3.14, z;
char *s1 = "hello";
char *s2 = "world";

z = max<double>( i, x);   // ok, max(T,S) returns 3.14
k = max( i, j);           // ok, max(T,T)
cout << max( s1, s2);     // ok, "world"
```

Template classes

- All member functions are template
- Lazy instantiation
- Possibility of partial specialization
- Specialization may completely different
- Default parameters are allowed

Dependent types

- Until type parameter is given, we are not sure on member
- Specialization can change
- If we mean type: typename keyword should be used

```
long ptr;
template <typename T>
class MyClass
{
   T::SubType * ptr;   // declaration or multiplication?
   //...
};
template <typename T>
class MyClass
{
   typename T::SubType * ptr;
   //...
};

typename T::const_iterator pos;
```

Two phase lookup

- There is two phases for template parse and name lookup

```
void bar()
{
  std::cout << "::bar()" << std::endl;
}

template <typename T>
class Base
{
public:
    void bar() { std::cout << "Base::bar()" << std::endl; }
};

template <typename T>
class Derived : public Base<T>
{
public:
    void foo() { bar(); }  // calls external bar() or error !!
};
```

Two phase lookup

- There is two phases for template parse and name lookup

```
void bar()
{
  std::cout << "::bar()" << std::endl;
}

template <typename T>
class Base
{
public:
    void bar() { std::cout << "Base::bar()" << std::endl; }
};

template <typename T>
class Derived : public Base<T>
{
public:
    void foo() { this->bar(); }   // or Base::bar()
};
```

Mixins

- Class inheriting from its own template parameter
- Not to mix with other mixins (e.g. Scala)

```
template <class Base>
class Mixin : public Base  { ... };

class RealBase { ... };
Mixin<RealBase> rbm;

//  be careful with lazy instantiation
template <class Sortable>
class WontWork : public Sortable
{
public:
    void sort()
    {
        Sortable::srot();  // !!misspelled
    }
};
WontWork<HasSortNotSrot> w;  // still compiles
w.sort()   // syntax error only here!
```

Liskov substitutional principle

- Barbara Liskov 1977: object of subtype can replace object of base
- Mixin<Derived> is not inherited from Mixin<Base>

```
class Base { ... };
class Derived : public base { ... };

template <class T> class Mixin : public T { ... };

Base        b;
Derived     d;

Mixin<Base>     mb;
Mixin<Derived>  md;

b  = d      // OK
mb = md;    // Error!
```

Curiously Recurring Template Pattern (CRTP)

- James Coplien
- Static polymorphism

```
template <typename T>
struct Base
{
    // ...
};

struct Derived : Base<Derived>
{
    // ...
};
```

Static polymorphism

- When we separate interface and implementation
- But no run-time variation between objects

```
template <class Derived>
struct Base
{
    void interface()
    {
         // ...
         static_cast<Derived*>(this)->implementation();
         // ...
    }
};
struct Derived : Base<Derived>
{
     void implementation();
};

std::vector<Derived>  vec;
for(auto d : vec)  d.interface();
```

Using (C++11)

- Typedef won't work well with templates
- Using introduce type alias

```
using myint = int;
template <class T> using ptr_t = T*;

void f(int) { }
// void f(myint) { }   syntax error: redeclaration of f(int)

// make mystring one parameter template
template <class CharT> using mystring =
    std::basic_string<CharT,std::char_traits<CharT>>;
```

Variadic templates (C++11)

- Type pack defines sequence of type parameters
- Recursive processing of pack

```
template<typename T>
T sum(T v)
{
  return v;
}
template<typename T, typename... Args>  // template parameter pack
T sum(T first, Args... args)            // function parameter pack
{
  return first + sum(args...);
}

int main()
{
  long lsum = sum(1, 2, 3, 8, 7);

  std::string s1 = "x", s2 = "aa", s3 = "bb", s4 = "yy";
  std::string ssum = sum(s1, s2, s3, s4);
}
```

Mixin reloded (C++11)

- Variadic templates make us possible to define variadic set of base

```
struct A {};
struct B {};
struct C {};
struct D {};

template<class... Mixins>
class X : public Mixins...
{
public:
    X(const Mixins&... mixins) : Mixins(mixins)... { }
};

int main()
{
  A a; B b; C c; D d;

  X<A,B,C,D> xx(a,b,c,d);
}
```

Template metaprograms

- First template metaprogram: Erwin Unruh 1994
  - Printed prime numbers as compiler error messages
- Later proved to be a Turing-complete sublanguage of C++
- Today many use cases
  - Expression templates
  - DSL (boost::xpressive)
  - Generators (boost::spirit)
  - Compile-time adaptation (std::enable_if)
- Many more...

Factorial

- Compile time recursion
- Specialization to stop recursion

```
template <int N>
struct Factorial
{
    enum { value = N * Factorial<N-1>::value };
};
template <>
struct Factorial<1>
{
    enum { value = 1 };
};

// example use
int main()
{
    const int fact5 = Factorial<5>::value;
    std::cout << fact5 << endl;
    return 0;
}
```

Meta-control structures

```
template <bool condition, class Then, class Else>
struct IF
{
    typedef Then RET;
};

template <class Then, class Else>
struct IF<false, Then, Else>
{
    typedef Else RET;
};

// example use
template <typename T, typename S>
IF< sizeof(T)<sizeof(S), S, T>::RET max(T t, S s)
{
    if ( t < s)
        return s;
    else
        return t;
}
```

Template metafunction as parameter

```
template <int n, template<int> class F>
struct Accumulate
{
    enum { RET = Accumulate<n-1,F>::RET + F<n>::RET };
};

template <template<int> class F>
struct Accumulate<0,F>
{
    enum { RET = F<0>::RET };
};

template <int n>
struct Square
{
    enum { RET = n*n };
};

cout << Accumulate<3,Square>::RET << endl;
```

Why use it?

```
template <unsigned long N>
struct binary
{
    static unsigned const value = binary<N/10>::value * 2 + N % 10;
};

template <>
struct binary<0>
{
    static unsigned const value = 0;
};

int main()
{
    const unsigned int di = 12;
    const unsigned int oi = 014;
    const unsigned int hi = 0xc;

    const unsigned int bi0 = binary_value("1101");  // run-time
    const unsigned int bi1 = binary<1100>::value;   // compile-time
}
```

Use case example

```
template <class T>
class matrix
{
public:
    matrix( int i, int j );
    matrix( const matrix &other);
    ~matrix();
    matrix operator=( const matrix &other);
private:
    int  x;
    int  y;
    T   *v;
    void copy( const matrix &other);
    void check( int i, int j) const throw(indexError);
};
```

Specialization for POD types

```
template <class T>
void matrix<T>::copy( const matrix &other)
{
    x = other.x;
    y = other.y;
    v = new T[x*y];
    for ( int i = 0; i < x*y; ++i )
       v[i] = other.v[i];
}

// specialization for POD types
template <>
void matrix<long>::copy( const matrix &other)
{
    x = other.x;
    y = other.y;
    v = new long[x*y];
    memcpy( v, other.v, sizeof(long)*x*y);
}

template <>
void matrix<int>::copy( const matrix &other) ...
```

Trait

```
template <typename T> struct is_pod {  enum { value = false }; };
template <> struct is_pod<long>     {  enum { value = true }; };

template <typename T, bool B> struct copy_trait
{
    static void copy( T* to, const T* from, int n) {
        for( int i = 0; i < n; ++i )
            to[i] = from[i];
    }
};
template <typename T> struct copy_trait<T, true>
{
    static void copy( T* to, const T* from, int n)    {
        memcpy( to, from, n*sizeof(T));
    }
};

template <class T, class Cpy = copy_trait<T,is_pod<T>::value> >
class matrix  { ... }
```

Policy

```
template <typename T> struct copy_trait
{
    static void copy( T* to, const T* from, int n)  {
        for( int i = 0; i < n; ++i ) to[i] = from[i];
    }
};
template <> struct copy_trait<long>
{
    static void copy( long* to, const long* from, int n) {
        memcpy( to, from, n*sizeof(long));
    }
};
template <class T, class Cpy = copy_trait<T> >
class matrix {  ... }

template <class T, class Cpy>
void matrix<T,Cpy>::copy( const matrix &other) {
    x = other.x;
    y = other.y;
    v = new T[x*y];
    Cpy::copy( v, other.v, x*y);
}
```

Typelist

```
class NullType {};

// We now can construct a null-terminated list of typenames:
typedef Typelist< char,
           Typelist<signed char,
             Typelist<unsigned char, NullType>
           >
        > Charlist;

// For the easy maintenance, precompiler macros are defined
// to create Typelists:

#define TYPELIST_1(x)              Typelist< x, NullType>
#define TYPELIST_2(x, y)           Typelist< x, TYPELIST_1(y)>
#define TYPELIST_3(x, y, z)        Typelist< x, TYPELIST_2(y,z)>
#define TYPELIST_4(x, y, z, w)  Typelist< x, TYPELIST_3(y,z,w)>

// usage example
typedef TYPELIST_3(char, signed char, unsigned char)  Charlist;
```

Typelist operations

```
//  Length
template <class TList> struct Length;

template <>
struct Length<NullType>
{
    enum { value = 0 };
};

template <class T, class U>
struct Length <Typelist<T,U> >
{
    enum { value = 1 + Length<U>::value };
};

static const int len = Length<Charlist>::value;
```

Typelist operations

```
// IndexOf
template <class TList, class T> struct IndexOf;

template <class T>
struct IndexOf< NullType, T>
{
    enum { value = -1 };
};

template <class T, class Tail>
struct IndexOf< Typelist<Head, Tail>, T>
{
private:
    enum { temp = IndexOf<Tail, T>::value };
public:
    enum { value = (temp == -1) ? -1 : 1+temp };
};

static const int IndexOf<Charlist, int>::value;            // -1
static const int IndexOf<Charlist, char>::value;           //  0
static const int IndexOf<Charlist, unsigned char>::value;  //  2
```

Matrix revisited

```
typedef TYPELIST_4(char, signed char, unsigned char, int)  Pod_types;

template <typename T> struct is_pod
{
    enum { value = IndexOf<Pod_types,T>::value != -1 };
};
template <typename T, bool B> struct copy_trait
{
    static void copy( T* to, const T* from, int n)   {
        for( int i = 0; i < n; ++i )
            to[i] = from[i];
    }
};
template <typename T> struct copy_trait<T, true>
{
    static void copy( T* to, const T* from, int n)   {
        memcpy( to, from, n*sizeof(T));
    }
};
template <class T, class Cpy = copy_trait<T,is_pod<T>::value> >
class matrix { ... }
```

Thank you!

gsd@elte.hu
http://gsd.web.elte.hu