165

Chapter 6

The C++ Programming Language

The Role of Classes

166

C++ as Enhanced C

- Stronger typing.
- Inline functions.
- Call by reference.
- Function prototypes and overloading.
- Default function parameters.
- Namespaces.
- Type-safe I/O facilities.
- High Level Programming (the Standard Template Library).

167

Strong Typing

- Many type-conversions are unsafe,
  but most of them are given for free by C.
- C++ tries to prevent unsafe type-conversions.
- The usage of casting is discouraged, and is
  considered a safety problem (and a bad taste).
  - So is the usage of type `void*`
- C++ provides special purpose casting operators,
  that are much safer, for rare occasions.

We'll meet them in the future

```
int i;
double x;

x = i;   // o.k.   (usually)
i = x;   // warning (usually)
```

```
int  *pi;
void *ptr;

ptr = pi;    // o.k.
pi  = ptr;   // syntax error !!
```

168

Strong Typing (const)

- A special treatment is given to `const` objects:
  - They cannot be modified.
    - » hence, they must be initialized.
  - They cannot be sent to a function that MAY modify them.
    - » Hence, a function must declare whether it modifies an argument.
    - » Note: An argument that is passed "by value" is never modified.

```
      char *str = "abc";   //   o.k  (for a transition period)
char const *cst = "abc";   //   o.k. – the only safe way
const int    ci = 15;      //   initialization
void func(int *);          //   a function declaration
func(&ci);                 //   syntax error at a function call
```

Will be a syntax error

May be a A warning

- Enumerated constants and `const` objects
  replace the need for MACROs that `#define` constants.

169

(The Evil of) MACROs

MACROs:

- **Advantages:**
  - Global constants
  - Common functionality without function call overhead
  - Common functionality on variety of types
- **Disadvantages:**
  - Do not obey scope rules
  - Type-unsafe constructs

C++ provides substitutes for MACROs without the above disadvantages:

`enum` and `const` types (with strong typing) - to substitute MACRO constants

`inline` functions - to eliminate function call overhead

`template` functions - for type-safe common functionality on different types

170

Inline Functions

The following function will not incur function call overhead, when called

```
inline int
max(int a, int b)
{ return a < b ? b : a; }
```

The function definition should appear in each file where it is used

- Hence, it is, usually, written on a header-file, that is `#included` wherever needed
- In fact, the compiler is not obliged to substitute the function call by its body but it will, if you use this feature judiciously (no recursive functions, please ;-)

This is a first step toward eliminating the need for an analogue MACRO

- Indeed, it is not useful for evaluating the `max()` of two `double` objects
- Nevertheless, the MACRO is not fool-proof either (`max()` of two `char*` objects ?)
- The full elimination of the need for MACROs is achieved by using `template` functions

templates will be taught later in this course

171

Initializations of Objects

- In C, a new (automatic) object can be declared at the top of a block only*, when its intended value is yet unknown - hence, left uninitialized
  (*) This is no longer true in C99
- Many bugs are related to uninitialized objects.
- C++ allows to declare objects at the middle of a block and even at `if` and `loop` headers
  - The scope of the latter is just the `if` or `loop` statement

```
Set s = SetCreate();
if (s == NULL) return FAILURE;
Result r = SetAdd(s,el);
for (Element e = SetFirst(s); e != NULL; e = SetNext(s)) {
  // do something with e
}
e = el;          //   syntax error: e does not exist
```

172

Call by Reference

```
void
swap(int& a, int& b)
{ int t = a; a = b; b = t;
}

swap(x,y); // OK
swap(5,y); // Error


int
sum(const large_data&);


double&
index(double a[], int indx)
{ return a[indx]; }

y = index(vec,j);     // same as y = vec[j];
index(vec,i) = 5.2;   // same as vec[i]=5.2;
```

- t& is a memory location of type t.
- Can serve as a lhs (left-hand-side) in an assignment operation.
- An expression of type t& can serve anywhere an expression of type t can, but not vice-versa.
- Used when value of an argument is to be modified.
- Also used for efficiency, as only address of location (a pointer) is passed, not value itself.
- The keyword *const* is used to prevent modifying the object.

173

Call by Reference (Cont.)

Interesting facts about references

- Reference objects **must be initialized**, by referring to a specific object, and cannot refer to any other object during their lifetime.
- A **temporary** object may be referred to by a `const` **reference only**.
- A function that returns a reference to an object should **guarantee** that the **referenced object**, that is returned, **exists**.

```
complex&
Add(complex& a, complex& b)
{ complex x = a;
  x.r += b.r;
  x.i += b.i;
  return x;   // A BUG
}
```

174

Function Prototyping and Overloading

```
void swap(int&,int&);
void swap(double&,double&);

int a,b;
double c,d;

swap(a,b);   // calls first swap
swap(c,d);   // calls second swap
swap(a,c);   // syntax error
```

- A function is identified by its **entire prototype** (signature), not just by its name.
- The function whose entire prototype **matches best** the calling statement is invoked (**if exists**).

175

Default Function Parameters

```
typedef struct {
   double re,im;
} Complex;
```

- Function parameters may be given default values.

```
Complex ComplexCreate(double r = 0., double i = 0.)
{  Complex c;  // re and im are undefined.
   c.re = r;
   c.im = i;
   return c;   // return a copy of c.
}
```

- The default is used if the parameter is not supplied.

```
Complex c;     // re and im are undefined
Complex c1 = ComplexCreate(2,3);
Complex c2 = ComplexCreate(2,0);
Complex c3 = ComplexCreate(2);   // same as (2,0)
Complex c4 = ComplexCreate();    // same as (0,0)
```

176

Namespaces

Suppose we have two versions of the function `sqrt()`,
one is implemented for speed and the other – for accuracy.
Furthermore, we have a huge software that uses one of them,
and we want to switch to the other version. In C, we probably have
them in two libraries, and we will link the software with the other library.
But then we may get a version of `pow()` that we do not want.
Moreover, what are we to do if we need both versions in the same software?

```
namespace std {
  double sqrt(double);
  double pow (double, double);
}
```

```
namespace alt {
  double sqrt(double);
  double pow (double, double);
}
```

```
double pyth(double x, double y)
{ return std::sqrt(x*x + y*y);
}
```

```
double PDE_solver(PDE e)
{ ...
  var1 = alt::sqrt(var2);
  ...
}
```


177

Namespaces (Cont.)

- Namespaces can be nested.
- Namespaces can be extended.

```
namespace  B {  ...  }
```

```
namespace  A {  ...  }
namespace  B {  ...  }
namespace  A {  ...  }
```

- Global access to a selected identifier in a namespace is achieved by a using declaration.

```
using alt::sqrt;
```

- Global access to all names in a namespace is achieved by a using directive. (*using directives* should be used judiciously and **never** in headers)

```
using namespace std;
```

- Identifiers within an unnamed namespace are available in the same translation unit only.

Ideally, namespaces should
- Express a logical coherent set of features
- Prevent user access to unrelated features
- Impose minimal notational burden on users

(B.S. C++PL 3rd)

178

Input/Output with C++ (Not exactly enhanced C)

C

```
#include <stdio.h>
int    i;
double f;
. . .
scanf("%d %lf", &i, &f);
printf("%d %f\n", i, f);
fprintf(stderr, "%f", f);
```

++

```
#include <iostream>
using namespace std;
int i;
double f;
cin  >> i >> f;
cout << i << f << endl;
cerr << f;
```

cin, cout, cerr are objects which represent the stdin, stdout, stderr respectively.

Using operator >> and operator << objects can perform input/output.

operations are performed according to the types of the objects - no formats.

179

High Level Programming

- The main strategy of quality programming is
  Reuse, Reuse, Reuse …
- One of the more common tasks in software is
  the usage of Containers
- Using containers requires the usage of
  Iterators and Algorithms
- C++ provides in its standard library self resizable containers, iterators and common algorithms. It is made in the style of Generic Programming and is implemented by using C++ Templates.
- In short, this is now known as STL, the
  Standard Template Library

180

High Level Programming (cont.)

- A special type of container is `string`, that may contain different types of character as needed. It eliminates the need for using raw C-style character arrays and pointers.
- Some of the containers are
  `vector, deque, list, set, map`
- Iterators may be `const` and/or `reverse`. There are also
  `istream_iterator, ostream_iterator`
- Among the algorithms there are
  `for_each, find, copy, replace, remove, reverse, sort, merge`

181

A Container Example

```
#include <iostream>
  using std::cin;
  using std::cout;
  using std::endl;

#include <string>
  using std::string;

#include <vector>
  using std::vector;

#include <list>
  using std::list;

#include <deque>
  using std::deque;

typedef  string         lmnt;      // or int
typedef  deque<lmnt>    container; // or list or vector
typedef  container::iterator iterator;
```

182

A Container Example (cont.)

```
int
main()
{ const int  len = 10;
  lmnt       l[len] =
          //     { 9, 15, 12, -5, 1};
                 { "abc", "xyz", "pqr", "st"};

  container  cn0, cn1(10);
  cn0.push_back (l[0]);
  cn0.push_back (l[1]);
  cn0.push_front(l[2]);          //  Illegal for "vector"
  cn0.push_front(l[3]);          //  Illegal for "vector"
  cn0.push_back (l[3]);

  for (iterator i = cn0.begin(); i != cn0.end(); ++i){
    cout << *i << endl;
  }

  return 0;
}
```


183

A Stream-Iterator Example

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

int main()
{ istream_iterator<int> is(cin);
  ostream_iterator<int> os(cout);
  for (int len = *is++; len-- > 0; ++is, ++os) {
    *os = *is;
    cout << ' ';
  }                              // An extra int may be read! However,
  if (cin.fail()) cin.clear();// printing int-as-string buys nothing
  istream_iterator<string> ist(cin);
  istream_iterator<string> end_of_stream;
  ostream_iterator<string> ost(cout);
  string space = ", ";
  for (; ist != end_of_stream ||(*ost = "\nFIN\n",0); ++ist, ++ost) {
      *ost = *ist;
    *++ost = space;
  }
  return 0;
}
```

184

An Algorithm Example

```
#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>
using namespace std;

typedef  int                  lmnt;
typedef  list<lmnt>           container;
typedef  container::iterator  iterator;
lmnt const target = 5;

int main()
{ istream_iterator<lmnt> is(cin), eos;
  container bucket;

  copy(is, eos, back_inserter(bucket));
  iterator it = find(bucket.begin(), bucket.end(), target);
  ostream_iterator<int> os(cout, ", ");
  os = copy(it, bucket.end(), os);
  cout << endl;
}
```


185

C++ as an OOP Language

- Classes.
- Operator Overloading.
- Templates.
- Inheritance - where OOP begins

186

Classes

- A class provides the means for the user to define objects, with associated functions and operators.

```
// An elementary implementation of type String in file my_string.h

#include <iostream.h>
#include <string.h>

const int maxLen = 255;

class String {
        char s[maxLen + 1];
        int  len;
    public:
        void assign(char const *st)
                          { strcpy(s,st); len = strlen(st); }
        int  length()     { return len; }
        void print()      { cout << s << endl; }
};
```

The old style

Using a const variable

data members

A hidden bug

member functions

- Objects are instances of classes:

```
String s1,s2;
```


187

The Class as an Enhanced C Structure

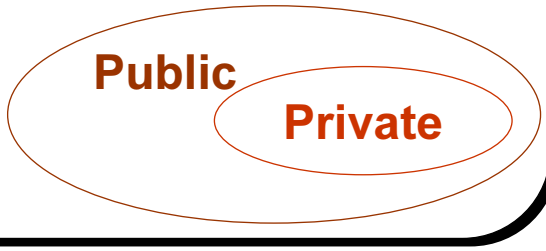
- **Additions to the C structure concept:**
  - **Member fields can be data, functions or operators.**
- **Enables information hiding.**
  - `public` **and** `private` **members.**
  - **Other access control features:**
    - `protected`
    - `friend`

We shall meet them in the future

188

Private

- Data hiding (one of the forms of encapsulation), can prevent unanticipated modifications to internal data structures.
- `private` members are accessible only by other member functions of the class.
- Exceptions of this rule are functions or classes explicitly listed as friends.
- `private` enables information hiding.



189

Public

- `public` indicates the visibility of the members that follow it.
- `public` members are accessible to any function within the scope of the class declaration.
- All class members are private by default.

`struct = class + public:`

190

Using Classes

```
#include <iostream.h>
#include "my_string.h"        // Testing the class String

static char const kg[] = "My name is Kurt Godel";

int main()
{
    String one, two; // Defining two objects.

    one.assign("My name is Alan Turing.");
    two.assign(kg);

    / / Print the shorter of one and two.

    if (one.length() <= two.length())
        one.print();
    else
        two.print();

    return 0;
}
```


191

Remarks about Classes

- Member functions have one "less" parameter than expected

```
int  length();      // declaration inside class
 one.length()       // usage with an object of the class
```

- Each member function has an implicit (hidden) parameter, which is an object of that class.
- At a function call, an argument for this parameter MUST be an object of the same class - no conversions are allowed.
- The name `this` points to this parameter - when needed.
- A member function may, and should, declare if it does not modify its hidden parameter (and our class declaration should be updated accordingly):

```
int  length() const      { return len; }
void print()  const      { cout << s << endl; }
```

192

Construction of Objects

The programmer may specify what happens when an object is "born", via a **constructor**:

```
#include <iostream>
#include <cstring>
using std::cout;
using std::endl;

class String {
  public:
    enum {maxLen = 255};
       // A constructor.
     String(char const *st)
                  { strcpy(s,st); len = strlen(st);}
     int  length() const { return len; }
     void print()  const { cout << s << endl; }
  private:
    char s[maxLen + 1];
    int len;
};
```

The new standard - 1997

A better way for integral constants

193

Using Constructors

```
#include <iostream.h>
#include "my_string.h"
static char const kg[] = "My name is Kurt Godel";

int main()
{
  String one("My name is Alan Turing");
  const String two(kg);

  // Print the shorter of one and two.
  if (one.length() <= two.length())
    one.print();
  else
    two.print();

  ....
  return 0;
}
```

If `print()` was not a **`const`** function we would have had a **syntax error** here

194

Calling a Constructor Explicitly

- **The constructor function may be called explicitly, returning an object of the class:**

The following are equivalent:

```
String one("abc");              // the constructor is applied to one

String one = String("abc");     // the constructor creates a
                                // temporary object, which is
                                // copied to one
```

- **The default constructor does nothing. It has no arguments.**
- **It is canceled when any other constructor is defined.**

195

Destruction of Objects

The programmer may specify what happens when an object "dies", via a **destructor**:

```
class String {
  public:
    String(char const *st) // A constructor
    { len = strlen(st);
      s = new char[len+1]; // Allocate space as needed
      strcpy(s,st);
    }

    ~String()          // A destructor
    { delete[] s;      // Free space
    }

    int length() const { return len; }
    void print() const { cout << s << endl; }
  private:
    int len;
    char *s;      // A pointer to variable length string
};
```

196

Using Destructors

It is very rare that one needs to explicitly call the destructor. In most cases an implicit call is automatically created.

```
static char const kg[] = "My name is Kurt Godel";

int main()
{
  String one("My name is Alan Turing");
  String two(kg); // Constructors are invoked.


  ....

  return 0; // one.~string() and two.~string()
            // are automatically called here
            // as both one and two go out of scope
}
```


197

Encapsulation

```
String s1("abc");

cout << s1.s;     // Error: Inner structure is private
s1.print();       // OK: Interface dictates the protocol
```

Scope

The `::` operator indicates **scope** (for either a class or a namespace).

Example:

```
String::print()
```

is the `print()` member of class `String`.

198

Defining Member Functions

Member functions may be defined in the body of the class declaration (*implicitly inline*):

```
class String {
  public:
    enum {maxLen = 255};
    void print() const
      { cout << s << endl;}
  private:
    char s[maxLen + 1];
};
```

Or may be defined separately:

```
class String {
 public:
  enum {maxLen = 255};
  void print() const;//Declaration
 private:
  char s[maxLen + 1];
};

  // Definition (note the scope)
inline void
String::print()const
{ cout << s << endl;
}
```

The String's max-length is retrieved outside the class using `String::maxLen`

Note the direct access to data-members (of `this`) from member functions.

199

Building Classes (wrong)

Suppose we want to build a class for complex numbers

The wrong way:

```
class Complex {
  public:
    double real, imag;
    Complex (double r = 0, double i = 0)
    { real = r;
      imag = i;
    }
};
```

With this setting, users access the data members directly:

```
#include  "Complex.h"
Complex c;
do_something_with(c.real);
```

200

Building Classes (right)

Exposing the inner structure of objects results in unstable code and maintenance hardship

The right way:

```
#include <cmath>
using std::sqrt;

class Complex {
  public:
    Complex (double r = 0, double i = 0)
        :real(r),imag(i) {}
    double real_part() const { return real;}
    double imag_part() const { return imag;}
    double abs()       const
      { return sqrt(real*real+imag*imag);}
  private:
    double real, imag;
};
```

The **private** part is not of real interest for the user

201

Using Classes

The *client* code will look like this:

```
#include  "Complex.h"

Complex c;
do_something_with(c.real_part());
```

Since `Complex::real_part()` is **`inline`**, there is no runtime function call overhead.

202

Modifying Classes

Suppose you want to reimplement the class `Complex` and use polar coordinates as the internal representation.

We have to make sure that no user's code will break

- **How about having to modify 1M lines of code ?**

The first version of the class will run into this trouble.

In contrast, the latter version provides what we need

- **Because implementation details are hidden behind a well defined interface, which is implementation independent.**

203

Modifying Classes

The class can now be rebuilt:

```
#include <cmath>
using std::sqrt; using std::atan2;
                    // Don't look for PI :-<
class Complex  {
 public:
   Complex (double re = 0,  double im = 0)
    : r(sqrt(re*re+im*im)), theta(arg(re,im)) {}
    double real_part() const { return r*cos(theta);
}
    double imag_part() const { return r*sin(theta);
}
    double abs()       const { return r;
 private:
    double r, theta;
    double arg(double x, double y) const
    { return !x ? (y >= 0 ? PI/2 : -(PI/2)) :
atan2(y,x);}
};
```

Not the final version

The client code remains the same (but must be recompiled).

Only because the members are `inline`.

204

Static Members

The function `Complex::arg()` above raises a few points:

- **Its action is unrelated to any specific `Complex` object, yet, as a member function, it has a `this`.**
- **Moreover, its usage is only when `this` is not completed.**
- **Suppose we want to make `Complex::arg()` `public`. It's unreasonable to execute `z.arg(x,y)` for evaluating an expression that is totally unrelated to `z`.**

Similar wonders may be raised regarding `PI` (assuming it is a part of the implementation of `Complex`):

- **If we need it as a data member, why in the world each Complex object should have its own copy of `PI` ?**

205

Static Members (Cont.)

C++ provides the capability of having `static` members in classes:

- **`static` data members are unique, and belong to the class.**
- **`static` member functions does not have `this` parameter, hence, can directly access only `static` data members.**
- **`static` member function can be called either using an object of the class, or using the scope operator.**

Here are the modifications (only) for the class `Complex`:

```
class Complex {
 public:
   static const double PI;
 private:
   static double arg(double x, double y)
   { return !x ? (y >= 0 ? PI/2 : -(PI/2)) : atan2(y,x);}
};
const double Complex::PI = 3.14159265358979323385;
```

Seems to be the only case where `public` data is o.k.

Cannot be `const`

Initialize in implementation

```
double x   = Complex::PI;              // Usage
double arg = Complex::arg(x,sqrt(2));  // Usage (if visible)
```

206