Chapter 6

The C++ Programming Language

The Role of Classes
C++ as Enhanced C

• Stronger typing.
• Scoped enum
• The type specifier `auto`
• [Inline functions – was adopted by C99].
• Call by reference.
• Function prototypes and overloading.
• Default function parameters.
• Namespaces.
• Type-safe I/O facilities.
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*` (except at very low level).
- C++ provides special purpose casting operators, that are much safer, for rare occasions.

```
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 !
```
Strong Typing \( (\text{const}) \)

- A special treatment is given to \texttt{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.
    - An argument that is passed “by value” is never modified by a function.

```c
char *str = "abc";  // Bad (str[0] = ‘x’ => UB)
char const *cst = "abc";  // o.k. – the only safe way
void func(char *);        // a function declaration
func(cst);                // syntax error at a function call
```

- Enumerated constants and \texttt{const} objects may replace many \texttt{#define} constants.

In C++11, and furthermore in C++14, \texttt{constexpr} hints to the compiler about values and functions that can be computed at compile-time, yielding constant values.
Strong Typing: enum Problems

```c
enum A { a, b, c};

int a = 3;  // Error: ‘a’ is redeclared (same as C)

int main()
{
    enum X { x, y, z};
    double a = 3.14;    // OK, inner scope (in both C and C++)
    double x = 2.71828; // Error: ‘x’ is redeclared (same as C)
    X x1 = 1; // Error: invalid conversion from ‘int’ (OK in C!)
    A c = z; // Error: invalid conversion from ‘X’ (OK in C!)
    int n = y; // OK: enum2int conversion (in both C and C++)
    return 0;
}
```

X and A are type-names

Copyright 1995 - 2000 C. Gotsman & 1995 - 2016 Y.M. Kimchi, Computer Science Dept. Technion
Strong Typing: Scoped `enum` (C++11)

enum class A { a, b, c}; // The {} is a ‘scope’

int a = 3; // OK

int main()
{
    enum class X { x, y, z};
    double a = 3.14; // OK, inner scope (in both C and C++)
    double x = 2.71828; // OK
    X x1 = 1; // Error: invalid conversion from ‘int’ to ‘X’
    A c = X::z; // Error: invalid conversion from ‘X’ to ‘A’
    int n = A::a; // Error: invalid conversion from ‘A’ to ‘int’
    return 0;
}

:: is the scope operator
Strong Typing: Type specifier `auto` (C++11)

Let the compiler find the exact type of an object

```cpp
Set my_set = SetCreate(...);
#define EXTRACT_LMNT(e) … // Like TO_INT(e)
SET_FOREACH(e, my_set) {
    auto lmnt = EXTRACT_LMNT(e);
}
auto n; // Error: missing initialization
auto sz = my_set.size(); // Is it int / unsigned / long unsigned ?

The type of `auto` is the type of the initializer, but under certain circumstances it is not what you want (or wrong)

```cpp
auto root = static_cast<float>(sqrt(5.0)); // prefer casting
```

Here, (float)sqrt(5.0) will also work

Scott Meyers: Effective Modern C++, It#6

A C style example

Finds the exact type: Less `typedef`, `const` awareness, and much more

C++ version of `SetSize(my_set)`
(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
Inline Functions

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

```c
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 `#include`ed 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
Initializations of Objects

• In C90, a new (automatic) object can be declared at the top of a block only*, when its intended value is yet unknown - hence, left uninitialized

  (*) C99 has followed C++ on this issue

• 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 (C99: for)
  – The scope of the latter is just the if or loop statement

```c
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
```
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.
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**.

```cpp
complex&
Add(complex& a, complex& b)
{
    complex x = a;
    x.r += b.r;
    x.i += b.i;
    **return x;**  // **A BUG**
}
```
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).
typedef struct {
    double re, im;
} Complex;

Complex ComplexCreate(double r = 0., double i = 0.);

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

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)

• Function parameters may be given default values at declaration.

• The default is used if the parameter is not supplied.
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?

```cpp
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);
    ...
}
```
Namespaces (Cont.)

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

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

- Global access to a selected identifier in a namespace is achieved by a **using declaration**.
- Global access to all names in a namespace is achieved by a **using directive**. (**using directives** should be used judiciously and **never** in headers)
- Identifiers within an unnamed namespace are available in the same translation unit only.

```cpp
using alt::sqrt;
using namespace std;
```

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)
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);
```

`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.

```cpp
#include <iostream>
using namespace std;
int i;
double f;
cin >> i >> f;
cout << i << f << endl;
cerr << f;
```
C++ as an OOP Language

- High Level Programming (the Standard Template Library)
- Classes
- Operator Overloading
- Templates
- Inheritance - where OOP begins

Later in this chapter
Subsequent three chapters
High Level Programming
How C++ makes programming easier
(If you know C++ :-)

• 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
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`, `array`, `forward_list`, `unordered_map` (C++11)

• 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`
A Container Example
(just a review)

#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 int lmnt;       // or string
typedef deque<lmnt> container; // or list or vector
typedef container::iterator iterator;
A Container Example  (cont.)

The interface of iterators is built to resemble pointers

```cpp
int main()
{
    int const len = 10; // May be initialized by len(10);
    int const 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;
}
```

Initialize

"NULL"

Step FWD

"Dereference"
Aggregates – initialize & iterate

```cpp
int main()
{
    int const len(10); // Same as initialization: int const len = 10;
    int l[len] = { 9, 15, 12, -5, 1},
                m{1,2,3,4,5}, n[len]{6,7,8,9};
    container cn0 = {1,2,3}, cn1({2,3,4}), cn2{5,6,7};

    for (int n: {9,8,7}) { // A range-based for loop
        cn0.push_back(n);
    }

    for (int n: m) {
        cn0.push_back(n);
    }

    for (int &m: cn0){ // Reference allows modifications
        m += 2;
    }

    for (int m: cn0){ // The modified elements
        cout << m << ','; // Prints: 3,4,5,11,10,9,3,4,5,6,7,
    }

    return 0;
}
```

**C++11**

**Completely eliminate the need for iterators**

**Reference allows modifications**

---

Copyright 1995 - 2000 C. Gotsman & 1995 - 2016 Y.M. Kimchi, Computer Science Dept. Technion
A Stream-Iterator Example

```cpp
#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 = ", "; // May be initialized by string space(" , ");
    for (; ist != end_of_stream || (*ost = "\nFIN\n", 0); ++ist, ++ost) {
        *ost = *ist;
        *++ost = space;
    }

    return 0;
}
```

Possible input for 3 integers:
```
3 9 22 17
```
#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;
}
C++ as an OOP Language

- High Level Programming (the Standard Template Library).
- Classes.
- Operator Overloading.
- Templates.
- Inheritance - where OOP begins
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; }
};

- Objects are instances of classes:

String s1, s2;
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
Private = Implementation Details

- 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.
Public = Interface

• **public** indicates the visibility of the members that follow it.

• **public** members are accessible by any code within the scope of the class declaration.

• All class members are **private** by default.

  \[
  \text{struct} = \text{class} + \text{public} \\
  \text{class} = \text{struct} + \text{private}
  \]

• Why C++ needs **private** members?
Using Classes

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

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

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;
}
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; }
  ```
Construction of Objects

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

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

class String {
public:
    enum {maxLen = 255};  // static int 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;
};
```

Since the 1998 standard

A better way for integral constants

Another way - Not only int
Using Constructors

```cpp
#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.
Calling a Constructor Explicitly

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

The following are equivalent in their effect (but the 2nd may fail – we’ll understand it later):

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

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

• The compiler supplies a default Ctor for a class with none.
• The default constructor does nothing. It has no arguments.
• It is canceled when any other constructor is defined.
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 the allocated space
      }

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

   private:
      int len;
      char *s; // A pointer to variable length string
};
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
}
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()    // The full name of the function

    is the print() member of class String.
Defining Member Functions

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

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

Or may be defined separately:

```cpp
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 accessed outside the class using `String::maxLen`

Note the direct access to data-members (of this) from member functions.
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);
Building Classes (better)

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

A better way:

```cpp
#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
Using Classes

The \textit{client} code will look like this:

\begin{quote}
\begin{verbatim}
#include "Complex.h"

Complex c;
do_something_with(c.real_part());
\end{verbatim}
\end{quote}

Since \texttt{Complex::real\_part()} is \texttt{inline}, there is no runtime function call overhead.
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.
Modifying Classes

The class can now be rebuilt:

```cpp
#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); }
};
```

The client code remains the same (\textit{but must be recompiled}).

Only because the members are \textit{inline}.}

Look for \textit{Cheshire Cat Pattern}

Not the final version
Building Classes (right)

Using accessors - *setters* and *getters* – are an indication of a bad design. Do not ask an object to give you some internal information – ask it to do some job for you. The right way:

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

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

The *private* part is not of real interest for the user.
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`?
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 do 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 != 0. ? (y >= 0 ? PI/2 : -(PI/2)) : atan2(y,x);
    }

    const double Complex::PI = 3.1415926535897932385;

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

C++98: Initialize in implementation
C++11: You may Initialize inside

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

Cannot be const