Advanced Programming

ACSE-5: Lecture 5 – Overview Slides

Adriana Paluszny

Review of Concepts

- Brief review [based on Bjarne Stroustrup’s slides]

Programming

- Why C++?
- Why software?
- Where is C++ used?
- Hello World program
- Computation & Linking
- What is programming?
- Integrated Development Environment

```
#include "std_lib_facilities.h " //header

int main() // where a C++ programs start
{
    cout << "Hello, world\n";// output
    keep_window_open();      // wait
    return 0;                // return success
}
```

Types

- Builtin types: int, double, bool, char
- Library types: string, complex
- Input and output
- Operators—“overloading”
- Variable names in C++
- Simple computations
- Literals
- Declaration & initialization
- Type safety
- Programming philosophy

```
// inch to cm and cm to inch conversion:
int main()
{
    const double cm_per_inch = 2.54;
    int val;
    char unit;
    while (cin >> val >> unit) {// keep
    reading
        if (unit == 'i')       // 'i' for inch
            cout << val << "in == “
                << val*cm_per_inch <<
    "cm\n";
        else if (unit == 'c')           // 'c' for
    cm
            cout << val << "cm == "
                << val/cm_per_inch <<
    "in\n";
        else
            return 0; // terminate on a
    “bad                                    //
    unit”, e.g. 'q'
    }
}
```

Computation

- Expressing computations
  - Correctly, simply, efficiently
  - Divide and conquer
  - Use abstractions
  - Organizing data, **vector**
- Language features
  - Expressions
    - Boolean operators (e.g. ||)
    - Short cut operators (e.g. +=)
  - Statements
  - Control flow
  - Functions
- Algorithms

```
// Eliminate the duplicate words; copying unique words
    vector<string> words;
    string s;
    while (cin>>s && s!= "quit")       words.push_back(s);
    sort(words.begin(), words.end());
    vector<string>w2;
    if (0<words.size()) {
                w2.push_back(words[0]);
                for (int i=1; i<words.size(); ++i)
                                if(words[i-1]!=words[i])
                                w2.push_back(words[i]);
     }
    cout<< "found " << words.size()-w2.size()
                <<  " duplicates\n";
    for (int i=0; i<w2.size(); ++i)
                 cout << w2[i] << "\n";
```

Errors

- Errors (“bugs”) are unavoidable in programming
  - Sources of errors?
  - Kinds of errors?
- Minimize errors
  - Organize code and data
  - Debugging
  - Testing
- Do error checking and produce reasonable messages
  - Input data validation
  - Function arguments
  - Pre/post conditions
- Manage your errors

```
int main()
{
    try
    {
        // …
    }
    catch (out_of_range&) {
        cerr << "oops – some vector "
                        " index out of  range\n";
    }
    catch (…) {
        cerr << "oops – some
    exception\n";
    }
    return 0;
}
```

Writing a Program

- Program a simple desk calculator
  - Process of repeatedly analyzing, designing, and implementing
- Strategy: start small and continually improve the code
- Use pseudo coding
- Program organization
  - Who calls who?
- Importance of feedback

```
double primary()            // Num or '(' Expr ')'
{
    Token t = get_token();
    switch (t.kind) {
    case '(':                                   // handle '(' expression ')'
    {       double d = expression();
            t = get_token();
            if (t.kind != ')') error("')'  expected");
            return d;
    }
    case '8': // '8' represents number "kind"
            return t.value; // return value
    default:
            error("primary expected");
    }
}
```

Functions

- Declarations and definitions
- Headers and the preprocessor
- Scope
  - Global, class, local, statement
- Functions
- Call
  - by value,
  - by reference, and
  - by **const** reference
- Namespaces
  - Qualification with :: and **using**

```
namespace Jack {//  in Jack's header file
        class Glob{ /*...*/ };
        class Widget{ /*...*/ };
}


#include "jack.h";      // this is in your code
#include "jill.h";    // so is this

void my_func(Jack::Widget p)
{       // OK, Jack's Widget class will not
        // clash with a different Widget
        // …
}
```

Classes

- User defined types
  - **class** and **struct**
  - **private** and **public** members
    - Interface
  - **const** members
  - constructors/destructor
  - operator overloading
  - Helper functions
  - Enumerations **enum**
- **Date** type

```
// simple Date (use Month type)
class Date {
public:
    enum Month {
            jan=1, feb, mar, apr, may, jun, jul,
            aug, sep, oct, nov, dec
    };
    Date(int y, Month m, int d);  // check for valid
                                  // date and initialize
    // …
private:
    int y;                        // year
    Month m;
    int d;                        // day
};

Date my_birthday(1950, 30, Date::dec);        // error:
                                  // 2nd argument not a Month
```

Streams

- Devices, device drivers, libraries, our code
- The stream model,
  - type safety, buffering
  - operators << and >>
- File types
  - Opening for input/output
  - Error handling
    - check the stream state
- Code logically separate actions as individual functions
- Parameterize functions
- Defining >> for **Date** type

```
struct Reading {                    // a temperature reading
        int hour;                   // hour after midnight [0:23]
        double temperature;
        Reading(int h, double t) :hour(h),          temperature(t) { }
};

string name;
cin >> name;
ifstream ist(name.c_str());
vector<Reading> temps;              // vector of readings
int hour;
double temperature;
while (ist >> hour >> temperature) {            // read
        if (hour < 0 || 23 <hour)
                error("hour out of range");
        temps.push_back(            Reading(hour,temperature) ); // store
}
```

Design Principles for Programming a Class Library

- Implement types used in the application domain
- Derived classes inherit from a few key abstractions
- Provide a minimum number of operations, access functions
- Use a consistent, regular style, appropriate naming
- Expose the interface only
  - encapsulation
- Virtual functions
  - dynamic dispatching

```
void Shape::draw() const
        // The real heart of class Shape
        // called by Window (only)
{
        Fl_Color oldc = fl_color();                    // save old color
           // there is no good portable way of
           // retrieving the current style (sigh!)
        fl_color(line_color.as_int()); // set color and
                                                    // style
        fl_line_style(ls.style(),ls.width());

        // call the appropriate draw_lines():
        draw_lines();  // a "virtual call"
                                                    // here is what is specific for a
                    // "derived class" is done

        fl_color(oldc);          // reset color to previous
        fl_line_style(0);        // (re)set style to default
}
```

Free Store

- Built vector type
- Pointer type
- The **new** operator to allocate objects on the free store (heap)
- Why use free store?
- Run-time memory organization
- Array indexing
- Memory leaks
- **void***
- Pointers vs references

```
class vector {
    int sz;                         // the size
    double* elem;// a pointer to the elements
public:
    // constructor (allocate elements):
    vector(int s) :sz(s), elem(new double[s]) { }
    // destructor (deallocate elements):
    ~vector() { delete[ ] elem; }
    // read access:
    double get(int n) { return elem[n]; }
    // write access:
    void set(int n, double v) { elem[n]=v; }
    // the current size:
    int size() const { return sz; }
};
vector v(10);
for (int i=0; i<v.size(); ++i) {
    v.set(i,i); cout << v.get(i) << ' ';
 }
```

Arrays

- Vector copy constructor
- Vector copy assignment
- Shallow and deep copy
- Arrays—avoid if possible
- Overloading [ ]
  - i.e. defining [] for **vector**

```
class vector {
    int sz;                     // the size
    double* elem;               // pointer to elements
public:
    // constructor:
    vector(int s) :sz(s), elem(new double[s]) { }
    // …
    // read and write access: return a reference:
    double& operator[ ](int n) { return elem[n]; }
};

vector v(10);
for (int i=0; i<v.size(); ++i) {// works and
                               // looks right!
    v[i] = i;                  // v[i] returns a
                               // reference to the i[th] element
    cout << v[i];
}
```

Vector

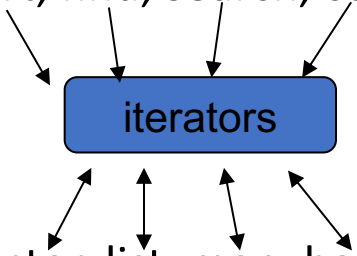
- Changing vector size
- Representation changed to include free *space*
- Added
  - **reserve(int n)**,
  - **resize(int n)**,
  - **push_back(double d)**
- The *this* pointer
- Optimized copy assignment
- Templates
- Range checking
- Exception handling

```
// an almost real vector of Ts:
template<class T> class vector { // "for all types T" int sz;                    // the size
        T* elem;        // a pointer to the elements
        int space;      // size+free_space
public:
        // default constructor:
        vector() : sz(0), elem(0), space(0);
        // constructor:
        explicit vector(int s)
                        :sz(s), elem(new T[s]), space(s) {
    // copy constructor:
        vector(const vector&);
        // copy assignment:
        vector& operator=(const vector&);
        ~vector() { delete[ ] elem; }           // destructor
        // access: return reference
        T& operator[ ] (int n) { return elem[n]; }
        int size() const { return sz; } // the current size

        // ...
};
```

The STL

- Generic programming
  - "lifting an algorithm"
- Standard Template Library
- 60 Algorithms
  - sort, find, search, copy, …



  - vector, list, map, hash_map,…
- 10 Containers
- iterators define a sequence
- Function objects

```
// Concrete STL-style code  for a more
// general version of summing values

template<class Iter, class T>          // Iter should be an
                                       // Input_iterator
                                       // T should be
                                       // something we can
                                       // + and =
T sum(Iter first, Iter last, T s)      // T is the
                                       // "accumulator type"
{
     while (first!=last) {
               s = s + *first;
               ++first;
    }
    return s;
}
```