CISC 3142
Programming Paradigms in C++

Ch2 – A Tour of C++:

The Basics, Procedural Programming

(*Stroustrup – The C++ Programming Language, 4th Ed*)

The Basics

- C++ has two kinds of entities
  - *Core language features*, such as built-in types, and loops
  - *Standard-library components*, such as containers, I/O operations
- Other than very few exceptions, the standard library is written in C++ itself, which attests to the language's expressiveness and efficiency
- C++ is a statically typed language – the type of every entity must be known to the compiler at its point of use

Type, Variables, and Arithmetic

- Fundamental data types:
  - bool, char, int, double, etc
  - Actual sizes can be obtained by the sizeof operator, sizeof(char) is 1
  - Note: while these are very similar to Java's primitive data types, types like short, int, long have sizes that are platform dependent.
- Initialization - the new {..} initializer lists disallow information loss

```
double d1 = 1.2;
double d2 {1.2};  // or double d2 = {1.2}; (= is optional with {…})
complex<double> z3 {1.2, 3.4};
int i1 = 5.6;
int i2 {5.6}; // error: information loss from float to int (narrowing conversion)
```

- const variables must be initialized at point of declaration

Type, Variables, and Arithmetic (cont')

- auto: when type can be deduced from the initializer
  - Used in C as a storage specifier for local variables, but C++ changed its meaning

```
auto b = false;          // bool
auto ch = 'a';           // char
auto z = sqrt(10);  // default to double
```

  - Avoid using auto when:
    - You want to make type clearly visible in a large scope
    - You want to be explicit about the type (e.g. sqrt(x) could be float or double)
  - Use auto when you want to avoid redundancy and writing long type names (common in generic programming)
- Arithmetic
  - The usual (x+=y, ++x, x%=y, etc)

Constants

- C++ has two keywords representing constants
  - const:
    - A promise for immutability and compiler will enforce the promise made by const
    - Primarily used with member function declaration to indicate that such function won’t change the state of the object
    - But const variables are still evaluated at runtime
  - constexpr:
    - “to be evaluated at compile time”
    - Primarily used for constants which will be put in special memory (not writable)
    - Performance is boosted since no need for runtime evaluation

Constants - examples

```
const int drinking_age = 21;                  // reassignment will trigger compiler error
int age = 13;                                 // reassign as you like
constexpr int dbl_age1 = 2*drinking_age;      // OK
constexpr int dbl_age2 = 2*age;               // error: age is not a const
double sum(const vector<double>&);            // sum() won't modify vector
vector<double> v {0.1, 1.2, 3.4, 5.6};        // v is not a constant
const double d1 = sum(v);                     // OK, const … can be evaluated at runtime
constexpr double d2 = sum(v);                 // error: sum(v) not constant
constexpr double square(double x) { return x*x;}  // a constexpr function
  // a constexpr function can be passed non-constant-expression arguments,
  // then no constant expression is returned, but that's OK – only 1 function is written
```

Pointers, Arrays, and Loops

- Pointers

```
char v[6] = ”Hello”;    // array of 6 characters: v[0] to v[5], v[5] is ’\0’, 6 is optional.
char* p;                // pointer to character
p = &v[1];              // p points to v[1], ’e’; & is the address-of operator; p[3]==’o’
char x = *p;            // x holds ’e’, * is the dereference operator (content-of)
```

For objects:

```
Thing t;                    // t is a value of Thing
Thing* p2t = &t;            // p2t now points to the same t
cout << p2t->name;          // ‘->’ allows member selection via a pointer, or
cout << (*p2t).name;        //  use the dereference to get the value
```

Using pointers as parameters can also allow functions such as swap() to handle (change) objects, the way references can too (since both pass the address)

Pointers, Arrays, and Loops (cont')

- Arrays and Looping

```
void print()
{
    int v[] = {0,1,2,3,4,5,6,7,8,9};      // size doesn't need to be specified
    for (auto x : v)                      // for each x in v, use auto& if elements are objects
        cout << x << '\n';
    int* p = &v[0];                       // p points to v[0]
    for (auto i=0; i<10; ++i)
        cout << *p++ << '\n';             // what if it's (*p)++?
    for (auto& x : v)                     // for each x, which is now a reference, see next slide
        cout << ++x << '\n';              // no need to dereference a reference
}
```

Pointers, Arrays, and Loops (cont')

- References
  - Implemented as a pointer, or an address, but no need for dereference
  - It can't be manipulated like a pointer (e.g. pointer arithmetic)
  - Must be initialized at declaration, can't be made to refer to a different object
  - Mainly used for passing parameters to functions – use const to prevent objects from being modified and accept passing literals
- Examples

```
int i = 1, j = 2;
int& ri = i;            // must be initialized right away, now ri is an alias of i
ri = j;                 // ri is still referring to i, not j, but now i has a value of 2
ri++;                   // i is now 3
cout << i << ", " << ri << '\n'; // prints: 3, 3
```

nullptr

- Marks no object to point to

```
double* pd = nullptr;
Link<Record>* lst = nullptr;
int x = nullptr;              // error: nullptr is a pointer not an integer
```

  - Before nullptr it used to be NULL which allows the above assignment (ambiguity)

  - If a function's paramter is a pointer, nullptr can be used to signify special case (such as the end of a list). In the following, the nullptr check avoids counting # of characters on nothingness

```
int count_x(char∗ p, char x) {      // count the number of occurrences of x in p[]
   if (p==nullptr) return 0;
   int count = 0;
   for (; ∗p!=0; ++p)               // initializer can be left out if we don't need it
      if (∗p==x)
         ++count;
   return count;
}
```

User-defined Types

- C++'s facility for *abstraction mechanisms*, from built-in types
- Structures (aggregate data only, usually no functions)

```
struct Vector {              // standard library has std::vector
    int sz;                  // number of elements
    double∗ elem;            // pointer to elements
};
```

Which can be initialized (all members of struct are public by default):

```
void vector_init(Vector& v, int s)
{
    v.elem = new double[s]; // allocate an array of s doubles
    v.sz = s;
}
```

Problem: user of Vector must know every detail of its representation

- Classes (a tighter connection between data and operations)

```
class Vector {
public: // default is private, but here we make interface public
    Vector(int s) :elem{new double[s]}, sz{s} { } // construct a Vector
    double& operator[](int i) { return elem[i]; } // element access: subscripting
    int size() { return sz; }
private: // data should be hidden
    double∗ elem; // pointer to the elements
    int sz;           // the number of elements
};
```

- Defining a type Vector:

```
Vector v(6); // values are not initialized yet
```

- v is really a handle with a fixed size itself (say 8 bytes), though the number of elements it represents can change, but elements are stored in the heap.
- And data are accessed only via interface: Vector(), operator[](), and size()
- no destructor yet (to release the memory obtained via new)

Enumerations

- Plain enums in C

  enum Traffic_light { red, yellow, green }; // we can use: Traffic_light lt1 = red;

  enum { arrow_up = 1, arrow_down}; // unnamed enums = a set of int constants

- C++ style

  enum class Traffic_light {red, yellow, green}; // strongly typed constants

  Traffic_light lt1 = Traffic_light::red; // must use scoped name, even in switch/case

  int i = Traffic_light::red; // error: not an integer

  Traffic_light lt2 = 0; // error: 0 is not a Traffic_light

- Java style

  enum MyFavColor {RED, YELLOW, GREEN}; // values in uppercase by convention

  - Colors must be referred to as MyFavColor.RED, except in switch/case (use RED)

Modularity

- The separation of declaration (interface) and definition (implementation)
- User code sees only declarations of types/functions used.
- The definitions of those types/functions are in separate source files and compiled separately.
  - To minimize compilation time
  - A library can be delivered in binaries, with the declarations in separate header files (.h, or .hpp) to be included in user code, so user can use the interface
- Namespaces: to avoid name clashes

  namespace My_names { …. }

  using namespace My_names; or, use My_names::variable_name

Error Handling

- Using built-in high-level constructs of data structures and algorithms can greatly limit chances for mistakes
- Exceptions
  - What if Vector's user try to access an out-of-range element?
  - The author of Vector doesn't know what is best for the user in this case
  - The user of Vector doesn't know where the problem may arise
  - The solution:
    - Let the author detect and tell the user about it (throws it)
    - The user then takes the appropriate action (catches it)
    - Proper unwinding of the call stack needs to be done by the implementation
  - Any object can be thrown in C++ (in Java, it must be Throwable)

Invariants

- One example of exception handling – a function checking its argument to ensure that it satisfies a precondition
- Vector example:

```
// author throws the exception
Vector::Vector(int s) {
    if (s<0) throw length_error{};
    elem = new double[s];
    sz = s;
}
```

```
// user catches/handles exception
try {
    Vector v(−27);
}  catch (std::length_error) {
    // handle negative size
}
```

- Invariants help us understand precisely what we want and force us to be specific, and should be properly enforced in constructors

Static Assertions

- Exceptions only report errors found at run time
- It's preferable to find error during compiler time if that's possible
- static_assert(A, S): prints S if A is not true

```
constexpr double C = 299792.458;                    // km/s
void f(double speed) {
    const double local_max = 160.0/(60∗60);     //160 km/h == 160.0/(60*60) km/s
    constexpr double local_max2 = 160.0/(60*60);
    static_assert(speed<C,"can't go that fast"); // error : speed must be a constant
    static_assert(local_max<C,"can't go that fast");    // error, local_max, as
   // a const, is evaluated at runtime. Compiler reports “non-constant condition for static
   // assertion”! Changing const to constexpr allows static_assert() to work.
    static_assert(local_max2<C, “can’t go that fast”); // OK
}
```

Chapter-end Advice

[1] Don’t panic! All will become clear in time; §2.1.

[2] You don’t have to know every detail of C++ to write good programs; §1.3.1.

[3] Focus on programming techniques, not on language features; §2.1.