CISC 3142
Programming Paradigms in C++

Ch6 – Type and Declarations

Ch7 – Pointers, Arrays and References

Basic Facilities: Elements of Imperative Programming

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

The ISO C++ Standard

- *Implementation-defined* behaviors (relates to hardware)

  unsigned char c1 = 64; // well defined: a char has at least 8 bits and can always hold 64

  unsigned char c2 = 1256; // implementation-defined: truncation if a char has only 8 bits

- *Undefined* behaviors
  - Behaviors are acceptable but implementer is not obliged to specify what occurs
    - Local variable usage before initialization
    - Integer division by zero (floating point division by zero is inf)
    - Buffer overflow (array out of bound)
    - Null pointer dereferencing
    - And more …
  - Allowing them affords performance gain by skipping checks

Types

- Fundamental Types
  - Boolean type (bool, 1/0 for true/false)
  - Character types (char, wchar_t)
  - Integer types (int, long long)
  - Floating-point types (double, long double)
  - void to signify the absence of information

  From the above, other types are constructed using declaratory operator (*, [], &)
  - Pointer types (int*)
  - Array types (char[])
  - Reference types (double&)

  User can define additional types:
  - Data structures and classes
  - Enumeration types for representing specific sets of values (enum class)

integral types

arithmetic types

built-in types

user-defined types

Exercises

```
bool b1 = -7;
int i1 = b1;
int i2 = i1 - true;
bool b2{nullptr};
bool b3 = i2 + b2;

char c1 = 255;
int i3 = c1;
void v;
void& rv = i3;
void* pv = &i3;
int i4 = *pv;
```

Declarations and definitions

- A definition is a declaration that supplies all that is needed for using the entity (such as setting aside the memory)
- Or declaration is the interface, and definition is the implementation
- Examples of declaration but no definition:

```
double sqrt(double); // function declaration
extern int error_number; // variable declaration
struct User; // type name declaration
```

- Except for function/namespace, a declaration is terminated by a semicolon
- Anatomy of a declaration

static	const char*	kings[]	=	{"Antigonus", "Seleucus", "Ptolemy"};
opt.prefix specifiers	base type	declarator name	opt. suffix function specifiers	optional initializer or function body

Exercises

```
const c = 7;
gt(int a, int b) { return (a>b) ? a : b;}
unsigned ui;
long li;
int* p, y; // int y, not int* y (operators apply to individual names only)
           // it's better to avoid this as it's prone to error
```

_identifier // Nonlocal names starting with an underscore are reserved for special facilities in the implementation and the run-time environment

__identifier, or _Identifier // double underscore, or _Uppercase are reserved

Scope

- A declared name can only be used in a specific part of the program text
  - *Local scope*: declared in function, inside a block defined by {}
  - *Class scope*: class member names
  - *Namespace scope*: in namespace outside any function
  - *Global scope*: outside any function, class, or namespace. Global namespace
  - *Statement scope*: within the () part of a for-, while-, if-, or switch-statement
  - *Function scope*: inside a function but outermost (i.e. >= local scope)
- Shadowed names are potential errors that would be hard to find
  - i.e. a name in a block can hide a declaration in an enclosing block or a global name. So using names such as i, x, for global variables is asking for trouble
  - A hidden global name can be referred to using scope resolution operator ::
- Variables can't be defined twice in the same scope

Initialization

- An initializer can use one of four syntactical styles:

  X a1 {v}; // since C+11, but preferred since it's less error-prone (no narrowing)
  X a2 = {v}; // supported since C (also no narrowing under C++)
  X a3 = v; // supported since C
  X a4(v); // calls constructor

- Some examples

  auto z1 {99}; // z1 is an std::initializer_list<int> when deduced, not int
  auto z2 = 99; // z2 is an int <- preferred when using auto
  vector<int> v1 {99}; // v1 is a vector of 1 element with the value 99
  vector<int> v2(99); // v2 is a vector of 99 0's **<- depends on constructor, {} is preferred**
  vector<string> v1{"hello!"}; // v1 is a vector of 1 element with the value "hello!"
  vector<string> v2("hello!"); // error : no vector constructor takes a string literal

- Empty initializer list (takes **default** values – determined by constructors for user-defined types)

  int x4 {}; // x4 becomes 0
  char* p {}; //p becomes nullptr
  vector<int> v4{}; // v4 becomes the empty vector
  string s4 {}; // s4 becomes "“
  Note: this also applies to (), i.e. Thing t = Thing(); == Thing t = Thing{}; == Thing t = {}; == Thing t{};

Missing Initializers

- Missing initializers is different from empty initializers
- The only good case for uninitialized variable is a large input buffer

```
constexpr int max = 1024*1024;
char buf[max]; // buffer will be overridden, so initialization may be wasted
some_stream.get(buf, max); // read at most max characters into buf
```

- Missing initializers will be the same as empty initializers in 4 cases
  - A **global**, **namespace**, **local static**, or **static member** is initialized with {}
  - Collectively they are called *static objects*
- Local variables and objects created on heap (*dynamic objects* or *heap objects*), are not initialized by default, unless they are of user-defined types with a default constructor

Examples

- **Missing initializers**

```
void f() {
  int x; // x does not have a well-defined value
  char buf[1024]; // buf[i] is not initialized
  int∗ p {new int}; //*p does not have a well-defined value
  char∗ q {new char[1024]}; // q[i] is not initialized
  string s; // s=="" because of string's default constructor
  vector<char> v; // v=={} because of vector's default ctor
  string∗ ps {new string}; // *ps is "" because of string's
                                      default constructor
  // ...
}
```

- **Empty and non-empty initializers**

```
void ff() {
  int x {}; // x becomes 0
  char buf[1024]{}; // buf[i] becomes 0 for all i
  int∗ p {new int{10}}; // *p becomes 10
  char∗ q {new char[1024]{}}; // q[i] becomes 0
                                        for all i
  // ...
}
```

Lifetimes of Objects

- The *lifetime* of an object
  - starts at the completion of its constructor
  - ends when its destructor starts executing
  - objects of types such as int, can be seen as having default do-nothing ctors/dtors
- Classifications
  - ***Automatic***: created and destroyed on stack as objects declared in a function
  - ***Static***: objects declared in global/namespace scope, and statics in functions/classes, live until the program ends. Also called *static* objects. Shared by all threads
  - ***Free store***: using the new and delete operators, lifetimes are directly controlled by the program
  - ***Temporary objects***: intermediate results in a computation, lifetime determined by their use. Typically temporary objects are automatic
  - ***Thread-local*** objects: declared thread_local, live and die with the thread
- *Static* and *automatic* are traditionally referred to as *storage classes*
- Array elements and nonstatic class members have the lifetimes determined by the object of which they are part

Type Aliases

- Give a new, more informative name for a type to avoid using an original name that's too long, complicated or ugly
- Examples

  using Pchar = char∗; // pointer to character

  using PF = int(∗)(double); // pointer to function taking a double and returning an int
- Old syntax uses keyword typedef

  typedef int int32_t; // equivalent to ''using int32_t = int;''

  typedef void(∗PtoF)(int); // equivalent to ''using PtoF = void(*)(int);''

Pointers

- The fundamental operation on a pointer is *dereferencing*, which is also called *indirection*.

```
char c = 'a';
char∗ p = &c; // p holds the address of c; & is the address-of operator
char c2 = ∗p; // c2 == ’a’; * is the dereference operator
```

- The smallest addressable object is a char (byte), can’t address bits
- More examples

```
int∗ pi; // pointer to int
char∗∗ ppc; // pointer to pointer to char
int∗ ap[15]; // array of 15 pointers to ints
int (∗fp)(char∗); // pointer to function taking a char* argument; returns an int
void∗ pv; // a pointer to an object of unknown type, must be explicitly converted to
another known type before dereferencing.
```

Arrays

- For a type T, T[size] is an array of size elements of type T
- Array can be accessed using subscript operator, [] or through a pointer
- Access out of the range is undefined and usually disastrous
  - In particular, runtime range checking is neither guaranteed nor common
- The size of the array must be a constant expression. Use vector for variable bounds

```
void f(int n) {
    int v1[n];            // error : array size not a constant expression (VLA)
    vector<int> v2(n);    // OK: vector with n int elements
}
```

- Multidimensional arrays are represented as arrays of arrays
- Array is the ideal solution for a simple fixed-length sequence of objects

More on arrays

- Three ways to allocate memory for arrays

```
int a1[10]; // 10 ints in static storage
void f() {
    int a2 [20]; // 20 ints on the stack
    int∗p = new int[40]; // 40 ints on the free store
}
```

- The built-in array is a low-level facility. There are also higher-level, better-behaved standard-library containers vector and array
- There is no array assignment, and name of an array implicitly converts to a pointer to its first element
- If array is allocated on heap, delete[] it once only, after last use
- If array is allocated statically or on stack, never delete[] it
- C-style string is a zero-terminated array of chars

Array Initializers

```
char v1[] = "ab"; // OK
char v2[] = { 'a', 'b', 0 }; // OK
char v3[3] = { 'a', 'b', 0 }; // OK
char v4[2] = { 'a', 'b', 0 }; // error : too many initializers

int v5[8] = { 1, 2, 3, 4 };
```

is equivalent to

```
int v5[] = { 1, 2, 3, 4 , 0, 0, 0, 0 }; // extra cells will be filled with 0
```

- No built-in copy operation for arrays – use vector/array/valarray instead

```
int v6[8] = v5; // error : can't copy an array (cannot assign an int* to an array)
v6 = v5; // error : no array assignment
```

String Literals

- A character sequence enclosed within double quotes
- It’s terminated by the null character, ’\0’, or, value 0

```
char greeting[10] = "Hello"; // Ok, greeting[5]~greeting[9] all filled with 0
sizeof(greeting) == 10; // true, though sizeof("Hello") == 6, i.e. counting the null
strlen(greeting) == 5; // true, strlen() is from <cstring> which doesn’t count null
```

- Don’t assign a string literal (stored in const memory) to a non-const char*:

```
void f() {
    char∗ p = "Plato"; // error, but accepted in pre-C++11-standard code
    p[4] = 'e'; // error : assignment to const
}
```

- To allow modification to a string, use non-const array. So replace the above with char p[] = "Plato"; which will allow elements in p to be modified

Pointers into Arrays

- Pointers and arrays are closely related in C++

```
int v[] = { 1, 2, 3, 4 };
int∗ p1 = v; // pointer to initial element (implicit conversion)
int∗ p2 = &v[0]; // pointer to initial element
int∗ p3 = v+4; // pointer to one-beyond-last element, DON'T dereference
```

- Navigating arrays (modern compilers should generate same code for the following)

```
void fi(char v[]) {
   for (int i = 0; v[i]!=0; ++i)
         use(v[i]);
}
```

```
void fp(char v[]) {
   for (char∗ p = v; ∗p!=0; ++p)
      use(∗p);
}
```

- Note the following equivalences: `a[j] == ∗(&a[0]+j) == ∗(a+j)`
- With `T* p`, note that the integer value of `p+1` will be `sizeof(T)` larger than that of `p`

Passing Arrays

- Arrays must be passed as a pointer to its first element

```
void comp(double arg[10]) { // arg is a double*, 10 is ignored here, element value not passed
    for (int i=0; i!=10; ++i)    // so the declaration is equivalent to void comp(double* arg)
        arg[i]+=99;
}
void f() {
    double a1[10];
    double a2[5];
    double a3[100];
    comp(a1); // Ok
    comp(a2); // disaster! Writes beyond bounds of a2
    comp(a3); // uses only the first 10 elements
};
```

Pointers and const

- Many objects don't have their values changed after initialization
  - Symbolic constants lead to more maintainable code than using literals
  - Many pointers are often read through but never written through
  - Most function parameters are read but not written to
- We use const to express immutability within scope after initialization

```
const int model = 90; // model is a const
const int v[] = { 1, 2, 3, 4 }; // v[i] is a const
const int x; // error : no initializer
```

- Note the scope of const

```
void g(const X∗ p) {
    // can't modify *p here
}
```

```
void h() {
    X val;      // val can be modified here
    g(&val); // not here due to g() definition
    // ...
}
```

References

- The main benefit of using a pointer vs. using a value (name of the object)
  - Passing potentially large amounts of data around at low cost
- The main differences vs values (they may be undesirable)
  - Different syntax: *p vs obj, p->m vs obj.m
  - A pointer can be reassigned to point to different objects at different times
  - Must be careful with pointers, they may be nullptr, or point to illegal places
- Can we have a solution addressing these differences? *Reference*!
  - Like a pointer, a reference is an alias for an object
  - It's usually implemented as memory address so it's not costlier than a pointer
- The differences between a reference and a pointer
  - You access a reference with exactly the same syntax as the name of an object
  - A reference always refers to the object to which it was initialized (no reseating)
  - There is no "null reference", and we always assume it refers to an object

Reference Usage

- References are mainly used for specifying arguments and return values for functions in general and for overloaded operators

```
template<class T>
class vector {
        T∗ elem;
        // …
public:
        T& operator[](int i) { return elem[i]; }                    // return reference to element
        const T& operator[](int i) const { return elem[i]; }        // return reference to const element
        void push_back(const T& a);                                  // pass element to be added by reference
        // …
};
void f(vector<double>& v) { // note: the book mistakenly used const vector<double>& v
        double d1 = v[1];      // copy the value of the double referred to by v.operator[](1) into d1
        v[2] = 7;              // place 7 in the double referred to by the result of v.operator[](2)
        v.push_back(d1);       // give push_back() a reference to d1 to work with
}
```

- There are 3 kinds of references: *lvalue* (mutable), const (immutable) and *rvalue* references (movable – whose value we don't need to preserve after we have used it)

Lvalue References

- X& means “reference to X”

```
void f() {
    int var = 1;
    int& r {var}; // r and var now refer to the same int
    int x = r; // x becomes 1 (r refers to the value)
    r = 2; // var becomes 2
    ++r; //  var is incremented to 1 (no operator operates on a reference so as to change it)
    int* pp = &r; // pp points to var
    int& r2; // error : initializer missing
    extern int& r3; // OK: r3 initialized elsewhere
    double& dr = 1; // error : lvalue needed on the right hand side
    const double& cdr {1}; // OK
}
```

- **Can’t have a pointer to a reference; can’t define an array of references**; i.e. a reference is not an object
- References are also commonly used as function argument so that the function can change the value of an object passed to it (refer to the swap() example covered before)
- References are also used as return types, so functions can be used both on lhs and rhs of an assignment

Pointers and References (Comparison)

- Use a pointer, if
  - you need to change which object to refer to
  - you can use =, +=, -=, ++, and – to change the value of a pointer variable
  - you want a collection of objects
  - you need a notion of “no value” (nullptr)
- Use a reference, if
  - you want to be sure that a name always refers to the same object
  - you want to pass a literal value (use const T&)

Comparison Examples

- Pointer arithmetic

```
void fp(char* p) {
    while (*p)
        cout << *p++;
}
void fr(char& r) {
    while (r)
        cout << r++;
// oops: increments the char referred
// to, not the reference, infinite loop!
}
// fix:
void fr2(char& r) {
    char* p = &r;
//get a pointer to object referred to
    while (*p)
        cout << *p++;
}
```

- Pointer for a collection of objects

```
int x, y;
string& a1[] = {x, y};
// error : array of references
string* a2[] = {&x, &y}; // OK
vector<string&> s1 = {x , y};
// error : vector of references
vector<string*> s2 = {&x, &y}; // OK
```

- Reference always refers to same object

```
template<class T> class Proxy {
// Proxy refers to the object with
// which it is initialized
    T& m;
public:
    Proxy(T& mm) :m{mm} {}
    // ...
};
template<class T> class Handle {
// Handle refers to its current object
    T* m;
public:
    Proxy(T* mm) :m{mm} {}
    void rebind(T* mm) { m = mm; }
    // ...
};
```