Notes for C++ Primer II

Lei Zhou

February 19, 2017

1 Variables and Basic Types

- C++ defines a set of primitive types that include the **arithmetic types** and a special type named **void**.
- The type of a string literal is ***array*** of constant `char`s. The compiler appends a null character ('\0') to every string literal. Thus, the actual size of a string literal is one more than its apparent size.
- Most generally, an object is a region of memory that can contain data and has a type.
- Initialization is not assignment. Initialization happens when a variable is given a value when it is created. Assignment obliterates an object's current value and replaces that value with a new one.
- The generalized use of curly braces for initialization was introduced as part of the new standard. This form of initialization previously had been allowed only in more restricted ways. This form of initialization is referred to as **list initialization**. Braced lists of initializers can now be used whenever we initialize an object and in some cases when we assign a new value to an object.
  When used with variables of built-in type, this form of initialization has one important property: The compiler will not let us list initialize variables of built-in type if the initializer might lead to the loss of information.
  ```
  long double ld = 3.1415926536;
  int a{ld}, b = {ld}; // error: narrowing conversion required
  int c(ld), d = ld; // ok: but value will be truncated
  ```
- Variables defined outside any function body are initialized to zero. With one exception, variables of built-in type defined inside a function are **uninitialized**. The value of an uninitialized variable of built-in type is undefined.
- A **declaration** makes a name known to the program. A file that wants to use a name defined elsewhere includes a declaration for that name. A **definition** creates the associated entity.
- To obtain a declaration that is not also a definition, we add the `extern` keyword and may not provide an explicit initializer.
  ```
  extern int i; // declares but does not define i
  int j; // declares and defines j
  ```
- A **scope** is a part of the program in which a name has a particular meaning.
- A **compound type** is a type that is defined in terms of another type.
  A declaration is a **base type** followed by a list of **declarators**. Each declarator names a variable and gives the variable a type that is related to the base type.
- A **reference** defines an alternative name for an object. A reference type "refers to" another type. We define a reference type by writing a declarator of the form `&d`, where `d` is the name being declared.
  Ordinarily, when we initialize a variable, the value of the initializer is copied into the object we are creating. When we define reference, instead of copying the initializer's value, we **bind** the reference to its initializer. Once initialized, a reference remains bound to its initial object. There is no way to rebind a reference to refer to a different object. Because there is no way to rebind a reference, reference *must* be initialized.
  We can define multiple references in a single definition. Each identifier that is a reference must be preceded by the `&` symbol.

- A **pointer** is a compound type that "points to" another type. Like reference, pointers are used for indirect access to other objects. Unlike a reference, a pointer is an object in its own right. Pointers can be assigned and copied; a single pointer can point to several different objects over its lifetime. Unlike a reference, a pointer need nit be initialized at the time it is defined. Like other built-in types, pointers defined at block scope have undefined value if they are not initialized.
  We define a pointer type by writing a declarator of the form `*d`, where `d` is the name being defined. The `*` must be repeated for each pointer variable.
  A pointer holds the address of another object. We get the address of an object by using the address-of operator (the **& operator**).
  When a pointer points to an object, we can use the dereference operator (the *** operator**) to access that object.

- Preprocessors variables are managed by the preprocessor, and are not part of the `std` namespace. Modern C++ programs generally should avoid using `NULL` and use `nullptr` instead.

- If possible, define a pointer only after the object to which it should point has been defined. If there is no object to bind to a pointer, then initialize the pointer to `nullptr` or zero. That way, The program can detect that the pointer does not point to an object.

- Two pointers are equal if they hold the same address and unequal otherwise.

- Generally, we use a `void*` pointer to deal with memory as memory, rather than using the pointer to access the object stored in that memory.

- It is a common misconception to think that the type modifier (`*` or `&`) applies to all the variables defined in a single statement.

- ```
  int i = 42;
  int *p; // p is a pointer to int
  int *&r = p; // r is a reference to the pointer p
  ```
  The easiest way to understand the type of r is to read the definition right to left. The symbol closest to the name of the variable (in this case the `&` in `&r`) is the one that has the most immediate effect on the variable's type.

- We can make a variable unchangeable by defining the variable's type as `const`.

- To define a single instance of a `const` variable, we use the keyword `extern` on both its definition and declaration(s).
  ```
  // file_1.cc defines and initializes a const that is accessible to other files
  extern const int bufSize = fcn();
  // file_1.h
  extern const int bufSize; // same bufSize as defined in file_1.cc
  ```

- As with any other object, we can bind a reference to an object of a `const` type. To do so we use a **reference to** `const`, which is a reference that refers to a const type;

- There are two exceptions to the rule that the type of a reference must match the type of the object to which it refers. The first exception is that we can initialize a reference to `const` from any expression that can be converted to the type of the reference. In particular, we can bind a reference to `const` to a non`const` object, a literal, or a more general expression.

- ```
  double dval = 3.14;
  const int &ri = dval;
  ```
  Here `ri` refers to an `int`. Operations on `ri` will be interger operations, but `dval` is a floating-point number, not an integer. To ensure that the object to which `ri` is bound is an `int`, the compiler transforms this code into something like
  ```
  const int temp = dval; // create a temporary const int from the double
  const int &ri = temp; // bind ri to that temporary
  ```
  In this case, `ri` is bound to a **temporary** object. A temporary object is an unnamed object created by the compiler when it needs a place to store a result from evaluation an expression.

- Like a reference to `const`, a **pointer to** `const` may not be used to change the object to which the pointer points.
- There are two exceptions to the rule that the types of a pointer and the object to which it points must match. The first exception is that we can use a pointer to `const` to point to a non`const` object.
  Like a reference to `const`, a point to `const` says nothing about whether the object to which the pointer points is `const`. Defining a pointer as a pointer to `const` affects only what we can do with the pointer. It is important to remember that there is no guarantee that an object pointed to by a pointer to `const` won't change.
- Unlike references, pointers are objects. Hence, as with any other type, we can have a pointer that is itself `const`. Like any other `const` object, a **const pointer** must be initialized, and once initialized, its value (i.e., the address that is holds) may not be changed. We indicate that the pointer is `const` by putting the `const` after the `*`.
- The fact that a pointer is itself `const` says nothing about whether we can use the pointer to change the underlying object.
- We use the term **top-level** `const` to indicate that the pointer itself is a `const`. When a pointer can point to a `const` object, we refer to that `const` as a **low-level** `const`.
- A **constant expression** is an expression whose value cannot change and that can be evaluated at compile time.
- Under the new standard, we can ask the compiler to verify that a variable is a constant expression by declaring the variable in a **constexpr** declaration.
- A **type alias** is a name that is a synonym for another type. We can define a type alias in one of two ways. Traditionally, we use a **typedef**:
  `typedef double wages; //` *wages is a synonym for double*
  The new standard introduced a second way to define a type alias, via an **alias declaration**:
  `using SI = Sales_item; //` *SI is a synonym for Sales_item*
- Under the new standard, we can let the compiler figure out the type for us by using the `auto` type specifier. The type that the compiler infers for `auto` is not always exactly the same as the initializer's type. Instead, the compiler adjusts the type to conform to normal initialization rules.
  First, when we use a reference as an initializer, the initializer is the corresponding object. The compiler uses that object's type for `auto`'s type deduction.
  Second, `auto` ordinarily ignores top-level `const`s. As usual in initializations, low-level `const`s, such as when an initializer is a pointer to `const`, are kept.
  When we ask for a reference to an `auto`-deduced type, top-level `const`s in the initializer are not ignored.
- Sometimes we want to define a variable with a type that the compiler deduces from an expression but do not want to use that expression to initialize the variable. For such cases, the new standard introduced a second type specifier, **decltype**, which returns the type of its operand. The compiler analyzes the expression to determine its type but does not evaluate the expression.
- As we've seen, when we dereference a pointer, we get the object to which the pointer points. Moreover, we can assign to that object. Thus, the type deduced by `decltype(*p)` is `int&`, not plain `int`.
- When we apply `decltype` to a variable without any parentheses, we get the type of that variable. If we wrap the variable's name in one or more sets of parentheses, the conpiler will evaluate the operand as an expression. A variable is an expression that can be the left-hand side of an assignment. As a result, `decltype` on such an expression yields a reference.
  `//` *decltype of a parenthesized variable is always a reference*
  `decltype((i)) d; //` *error: d is int& and must be initialized*
  `decltype(i) e; //` *ok: e is an (uninitialized) int*
- It is a common mistake among new programmers to forget the semicolon at the end of a class definition.
- Under the new standard, we can supply an **in-class initializer** for a data member. When we create objects, the in-class initializers will be used to initialize the data members.

- C++ programs also use the preprocessor to define **header guards**. Header guards rely on preprocessor variables.
  Headers should have guards, even if they aren't (yet) included by another header.