C++11 Knowledge Point

* Chapter 2
  + 2.4.1. References to const
  1. Terminology: const Reference is a Reference to const.Technically speaking, there are no const references.
  + 2.4.2. Pointers to const
  1. It may be helpful to think of pointers and references to const as pointers or references “that think they point or refer to const.”
  2. Like a references to const
  3. const pointers: pointer that is itself const.the address that it holds can not be changed.
  + 2.4.3. Top-Level const
  1. top-level const to indicate that the pointer itself is a const.
  2. pointer can point to a const object，we refer to that const as a low-level const.
  + 2.4.4. constexpr and Constant Expressions
  1. constant expression is an expression whose value cannot change and that can be evaluated at compile time.
  2. Literal Types:the arithmetic, reference, and pointer types are literal types
  3. Variables defined inside a function ordinarily are not stored at a fixed address Hence,we cannot use a constexpr pointer to point to such variables.On the other hand, the address of an object defined outside of any function is a constant expression.
  4. Pointers and constexpr: constexpr specifier applies to the pointer.

e.g. const int \*p = nullptr; constexpr int \*q = nullptr;

p is a pointer to const int,q is a const pointer to int.

* + 2.5.1. Type Aliases
  1. two ways:

typedef e.g.

typedef double wages; // wages is a synonym for double

typedef wages base, \*p; // base is a synonym for double, p for double\*

using e.g.

using SI = Sales\_item; // SI is a synonym for Sales\_item

* 1. Pointers, const, and Type Aliases

typedef char \*pstring;

const pstring cstr = 0; // cstr is a constant pointer to char

const pstring \*ps; // ps is a pointer to a constant pointer to char

The base type in these declarations is const pstring.

* + 2.5.2. The auto Type Specifier
    - 1. By implication ,a variable that uses auto as its type specifier must have an initializer.
      2. When we define multiple variables using auto,initializers for all the variables in the declaration must have types that are consistent with each other.
      3. The compiler adjusts the type to conform to normal initialization rules:

First: when we use reference,we are really use the object to which the reference refers.

Second: auto ordinarily ignores top-level consts; low-level consts, such as when an initializer is a pointer to const, are kept.

Third: if we want the deduced type to have a top-level const, we must say so explicitly.

e.g. const auto f = ci;// deduced type of ci is int; f has type const int

Forth: When we ask for a reference to an auto-deduced type, top- level consts in the initializer are not ignored. As usual, consts are not top-level when we bind areference to an initializer.

* + 2.5.3. The decltype Type Specifier
  1. returns the type of its operand. The compiler analyzes the expression to determine its type but does not evaluate the expression.
  2. The way decltype handles top-level const and references differs subtly from the way auto does.

When the expression to which we apply decltype is a variable, decltype returns the type of that variable,including top-level const and references.

* 1. decltype and References

When we apply decltype to an expression that is not a variable, we get the type that that expression yields.Generally speaking, decltype returns a reference type for expressions that yield objects that can stand on the left-hand side of the assignment

* 1. decltype of a parenthesized variable is always a reference(diff with auto)
  + 2.6.1. Defining the Sales\_data Type
  1. class begins with the keyword struct,the class body is surrounded by curly braces.The names defined inside the class must unique but can reuse names defined outside the class.
  2. The close curly that ends the class body must be followed by a semicolon.
  3. Class data members:

We define data members the same way that we define normal variables.

In-class initializer for a data member:Members without an initializer are default initialized.

* + 2.6.3. Writing Our Own Header Files
  1. Headers (usually) contain entities (such as class definitions and const and constexpr variables that can be defined only once in any given file.
  2. we need to write our headers in a way that is safe even if the header is included multiple times.(preprocesser)
  3. Whenever a header is updated, the source files that use that header must be recompiled to get the new or changed declarations.
  4. The preprocessor—which C++ inherits from C—is a program that runs before the compiler and changes the source text of our programs.

Header guards rely on preprocessor variables that have two state: defined and not defined. The #define directive takes a name and defines that name as a preprocesser variables.

#ifdef is true if the variable has been defined, and #ifndef is true if the variable has not been defined.

If the test is true, then everything following the #ifdef or #ifndef is processed up to the matching #endif.

* Chapter 3
  + 3.1. Namespace using Declarations

1. using declaration Format: using namespace::name; .There must be a using declaration for each name we use, and each declaration must end in a semicolon.
2. Headers Should Not Include using Declarations.
   * 3.2.1. Defining and Initializing strings
   1. default initialization,it's a empty string.
   2. string s1; string s2(s1); equivalent string s2 = s1; The s2 is a copy of s1.
   3. string s3(“value”) ; equivalent string s3 = “value”; The s3 is a copy of the string literal.
   4. string s4(n, “c”); Initialize s4 whit n copies of character 'c'.
   5. Direct and Copy Forms of Initialization.

When we initialize a variable using '=',we use copy initialization;when we omit '=',we use direct initialization. string s5 = “hiya”; //copy initialization

string s6(“hiya”); //direct initialization

string s7 = string(10, 'c');//indirectly copy initialization by explicitly creating a (temporary) object to copy. Not recommend.

* + 3.2.2. Operations on strings
  1. The string::size\_type Type,named companion type in order to use it in machine-independent manner.
  + 3.2.3. Dealing with the Characters in a string
  1. cctype
  + 3.3.1. Defining and Initializing vectors
  1. Templates are not themselves functions or classes.
  2. Some compilers may require the old-style declarations for a vector of vectors, for example, vector<vector<int> >.
  3. vector<string> v{10}; //v has ten default-initialized elements
  + 3.3.2. Adding Elements to a vector
  1. vectors Grow Efficiently.
  2. The body of a range for must not change the size of the sequence over which it is iterating.
  3. Subscript Only Elements that are Known to Exist!
  + 3.4.1. Using Iterators
  1. Terminology: Iterators and Iterator Types.Iterators is a set conceptually and gerneric types that they supports common actions. And iterator type supports the (conceptually) iterator.
  2. Combining Dereference and Member Access.Assume it is a iterator into the vector<string>.(\*it).empty() eq it->empty().
  + 3.4.2. Iterator Arithmetic(p.160)
  1. Operations Supported by only vector and string Iterators.
  2. The result type of subtract two iterators is signed integral type named different\_type.
  + 3.5.1. Defining and Initializing Built-in Arrays
  1. Array costs lot of flexibility to get the runtime performance advantage.
  2. Character Arrays Are Special.We can initialize such arrays from a string literal,notice:the string literal end with a null character '\0'.
  3. No Copy or Assignment
  4. Understand array declarations by starting with the array's name and reading them from the inside out then right left.
  + 3.5.2. Accessing the Elements of an Array
  1. define array use size\_t in cstddef head file.
  + 3.5.3. Pointers and Arrays