Variables

February 13, 2017

Each variable in C++ has a type. The type determines the size and layout of the variables memory, the range of values that can be stored within that memory, and the set of operations that can be applied to the variable. C++ programmers tend to refer to variables as “variables” or “objects” interchangeably.

1 Variable Definitions

A simple variable definition consists of a type specifier, followed by a list of one or more variable names separated by commas, and ends with a semicolon. Each name in the list has the type defined by the type specifier. A definition may (optionally) provide an initial value for one or more of the names it defines.

Most generally, an object is a region of memory that can contain data and has a type.

Initializers

An object that is initialized gets the specified value at the moment it is created. The values used to initialize a variable can be arbitrarily complicated expressions. When a definition defines two or more variables, the name of each object becomes visible immediately. Thus, it is possible to initialize a variable to the value of one defined earlier in the same definition.

double price = 109.99, discount = price * 0.16; // ok: price is defined and initialized before it is used to initialize discount

double salePrice = applyDiscount(price, discount); // ok: call applyDiscount and use the return value to initialize salePrice

Initialization and assignment are different operations in C++. Initialization is not assignment. Initialization happens when a variable is given a value when it is created. Assignment obliterates an objects current value and replaces that value with a new one.

List Initialization

The language defines several different forms of initialization. we can use any of the following four different ways to define an int variable named units_sold and initialize it to 0.

int units_sold = 0;

int units_sold = {0};

int units_sold{0};

int units_sold(0);

The generalized use of curly braces for 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

The compiler rejects the initializations of a and b because using a long double to initialize an int is likely to lose data. At a minimum, the fractional part of ld will be truncated. In addition, the integer part in ld might be too large to fit in an int.

2 const Qualifier

Define a variable whose value cannot be changed. We can make a variable unchangeable by defining the variables type as const.

const int bufSize = 512; // input buffer size

defines bufSize as a constant. Any attempt to assign to bufSize is an error:

bufSize = 512; // error: attempt to write to const object

Because we cant change the value of a const object after we create it, it must be initialized.

The initializer may be an arbitrarily complicated expression:

const int k; // error: k is uninitialized const

The type of an object defines the operations that can be performed by that object. A const type can use most but not all of the same operations as its nonconst version. Among the operations that don't change the value of an object is initialization—when we use an object to initialize another object, it doesnt matter whether either or both of the objects are consts.

int i = 42;

const int ci = i; // ok: the value in i is copied into ci int j = ci; // ok: the value in ci is copied into j

The constness of ci matters only for operations that might change ci. When we copy ci to initialize j, we don't care that ci is a const. Copying an object doesn't change that object. Once the copy is made, the new object has no further access to the original object.

By Default, const Objects Are Local to a File

When a const object is initialized from a compile-time constant, the compiler will usually replace uses of the variable with its corresponding value during compilation. To substitute the value for the variable, the compiler has to see the variable's initializer. In order to see the initializer, the variable must be defined in every file that wants to use the variables value. To support this usage, yet avoid multiple definitions of the same variable, const variables are defined as local to the file.

Sometimes we have a const variable that we want to share across multiple files but whose initializer is not a constant expression. We want to define the const in one file, and declare it in the other files that use that object. 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

To share a const object among multiple files, you must define the variable as extern.

2.1 References to const

Bind a reference to an object of a const type. A reference to const is a reference that refers to a const type. A reference to const cannot be used to change the object to which the reference is bound.

const int ci = 1024;

const int &r1 = ci; // ok: both reference and underlying object are const

r1 = 42; // error: r1 is a reference to const

int &r2 = ci; // error: non const reference to a const object

Because we cannot assign directly to ci, we also should not be able to use a reference to change ci. Therefore, the initialization of r2 is an error.

const Reference is a Reference to const. Technically speaking, there are no const references. A reference is not an object, so we cannot make a reference itself const. Because there is no way to make a reference refer to a different object, in some sense all references are const. Whether a reference refers to a const or nonconst type affects what we can do with that reference, not whether we can alter the binding of the reference itself.

3 Dealing with Types

3.1 Type Aliases

A type alias is a name that is a synonym for another type. We can define a type alias in one of two ways. One is typedef:

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

Declarations that include typedef define type aliases rather than variables. The declarators can include type modifiers that define compound types built from the base type of the definition.

The other is via an alias declaration:

using SI = Sales_item; // SI is a synonym for Sales_item

An alias declaration starts with the keyword using followed by the alias name and an =.

A type alias is a type name and can appear wherever a type name can appear.

wages hourly, weekly; // same as double hourly, weekly;

SI item; // same as Sales_item item

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

pstring is an alias for the the type char*. The base type in these declarations is const pstring. A const that appears in the base type modifies the given type. The type of pstring is "pointer to char". So, const pstring is a constant pointer to char—not a pointer to const char. It can be tempting, albeit incorrect, to interpret a declaration that uses a type alias by conceptually replacing the alias with its corresponding type.

const char *cstr = 0; // wrong interpretation of const pstring cstr

This interpretation is wrong. When we use pstring in a declaration, the base type of the declaration is a pointer type. When we rewrite the declaration using char*, the base type is char and the * is part of the declarator. In this case, const char is the base type. This rewrite declares cstr as a pointer to const char rather than as a const pointer to char.

3.2 The auto Type Specifier

It is not uncommon to want to store the value of an expression in a variable. To declare the variable, we have to know the type of that expression. When writing a program, it is difficult—sometimes even impossible—to determine the type of an expression. Let the compiler figure out the type by using the auto type specifier. auto tells the compiler to deduce the type from the initializer. A variable that uses auto as its type specifier must have an initializer.

// the type of item is deduced from the type of the result of adding val1 and val2

auto item = val1 + val2; // item initialized to the result of val1 + val2

The compiler deduce the type of item from the type returned by applying + to val1 and val2.

We can define multiple variables using auto. Because a declaration can involve only a single base type, the initializers for all the variables in the declaration must have types that are consistent with each other.

autoi=0,*p= &i; //ok: i is int and p is a pointer to int

auto sz = 0, pi = 3.14; // error: inconsistent types for sz and pi

When we use a reference, we are really using the object to which the reference refers. When we use a reference as an initializer, the initializer is the corresponding object. The compiler uses that objects type for autos type deduction.

int i = 0, &r = i;

auto a = r; // a is an int (r is an alias for i, which has type int)

Auto ordinarily ignores top-level consts, but low-level consts are kept.

const int ci = i, &cr = ci;

auto b = ci; // b is an int (top-level const in ci is dropped)

auto c = cr; // c is an int (cr is an alias for ci whose const is top-level)

auto d = &i; // d isan int*(& of an int object is int*)

auto e = &ci; // e is const int*(& of a const object is low-level const)

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

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

We can also specify that we want a reference to the auto-deduced type.

auto &g = ci; // g is a const int& that is bound to ci

auto &h = 42; // error: we can't bind a plain reference to a literal

const auto &j = 42; // ok: we can bind a const reference to a literal

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

When we define several variables in the same statement, it is important to remember that a reference or pointer is part of a particular declarator and not part of the base type for the declaration. The initializers must provide consistent auto-deduced types.

auto k = ci, &l = i; // k is int; l is int&

auto &m = ci, *p = &ci; // m is a const int&; p is a pointer to const int

auto &n = i, *p2 = &ci; // error: type deduced from i is int; type deduced from &ci is const int

3.3 The decltype Type Specifier

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. Decltype returns the type of its operand. The compiler analyzes the expression to determine its type but does not evaluate the expression.

decltype(f()) sum = x; // sum has whatever type f returns

The compiler does not call f, but it uses the type that such a call would return as the type for sum. The compiler gives sum the same type as the type that would be returned if we were to call f. 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.

const int ci = 0, &cj = ci;

decltype(ci) x = 0; // x has type const int

decltype(cj) y = x; // y has type const int& and is bound to x

decltype(cj) z; // error: z is a reference and must be initialized

cj is a reference, decltype(cj) is a reference type.

Decltype is the only context in which a variable defined as a reference is not treated as a synonym for the object to which it refers.

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

// decltype of an expression can be a reference type

int i = 42, *p = &i, &r = i;

decltype(r + 0) b; // ok: addition yields an int; b is an (uninitialized) int

decltype(*p) c; // error: c is int& and must be initialized

Here r is a reference, so decltype(r) is a reference type. If we want the type to which r refers, we can use r in an expression, such as r + 0, which is an expression that yields a value that has a nonreference type.

The dereference operator is an example of an expression for which decltype returns a reference. When we dereference a pointer, we get the object to which the pointer points. The deduction done by decltype depends on the form of its given expression. Enclosing the name of a variable in parentheses affects the type returned by decltype. When we apply decltype to a variable without any parentheses, we get the type of that variable. If we wrap the variables name in one or more sets of parentheses, the compiler 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

Decltype((variable)) (note, double parentheses) is always a reference type, but decltype(variable) is a reference type only if variable is a reference.