C++ Type System (Modern C++)

Visual Studio 2015

The new home for Visual Studio documentation is Visual Studio 2017 Documentation on docs.microsoft.com.

The latest version of this topic can be found at C++ Type System (Modern C++). The concept of *type* is very important in C++. Every variable, function argument, and function return value must have a type in order to be compiled. Also, every expression (including literal values) is implicitly given a type by the compiler before it is evaluated. Some examples of types include `int` to store integral values, `double` to store floating-point values (also known as *scalar* data types), or the Standard Library class std::basic_string to store text. You can create your own type by defining a `class` or `struct`. The type specifies the amount of memory that will be allocated for the variable (or expression result), the kinds of values that may be stored in that variable, how those values (as bit patterns) are interpreted, and the operations that can be performed on it. This article contains an informal overview of the major features of the C++ type system.

Terminology

Variable: The symbolic name of a quantity of data so that the name can be used to access the data it refers to throughout the scope of the code where it is defined. In C++, “variable” is generally used to refer to instances of scalar data types, whereas instances of other types are usually called “objects”.

Object: For simplicity and consistency, this article uses the term “object” to refer to any instance of a class or structure, and when it is used in the general sense includes all types, even scalar variables.

POD type (plain old data): This informal category of data types in C++ refers to types that are scalar (see the Fundamental types section) or are *POD classes*. A POD class has no static data members that aren’t also PODs, and has no user-defined constructors, user-defined destructors, or user-defined assignment operators. Also, a POD class has no virtual functions, no base class, and no private or protected non-static data members. POD types are often used for external data interchange, for example with a module written in the C language (which has POD types only).

Specifying variable and function types

C++ is a *strongly typed* language and it is also *statically-typed*; every object has a type and that type never changes (not to be confused with static data objects).
When you declare a variable in your code, you must either specify its type explicitly, or use the `auto` keyword to instruct the compiler to deduce the type from the initializer.
When you declare a function in your code, you must specify the type of each argument and its return value, or `void` if no value is returned by the function. The exception is when you are using function templates, which allow for arguments of arbitrary types.

After you first declare a variable, you cannot change its type at some later point. However, you can copy the variable’s value or a function’s return value into another variable of a different type. Such operations are called *type conversions*, which are sometimes necessary but are also potential sources of data loss or incorrectness.

When you declare a variable of POD type, we strongly recommend you initialize it, which means to give it an initial value. Until you initialize a variable, it has a "garbage" value that consists of whatever bits happened to be in that memory location

previously. This is an important aspect of C++ to remember, especially if you are coming from another language that handles initialization for you. When declaring a variable of non-POD class type, the constructor handles initialization.

The following example shows some simple variable declarations with some descriptions for each. The example also shows how the compiler uses type information to allow or disallow certain subsequent operations on the variable.

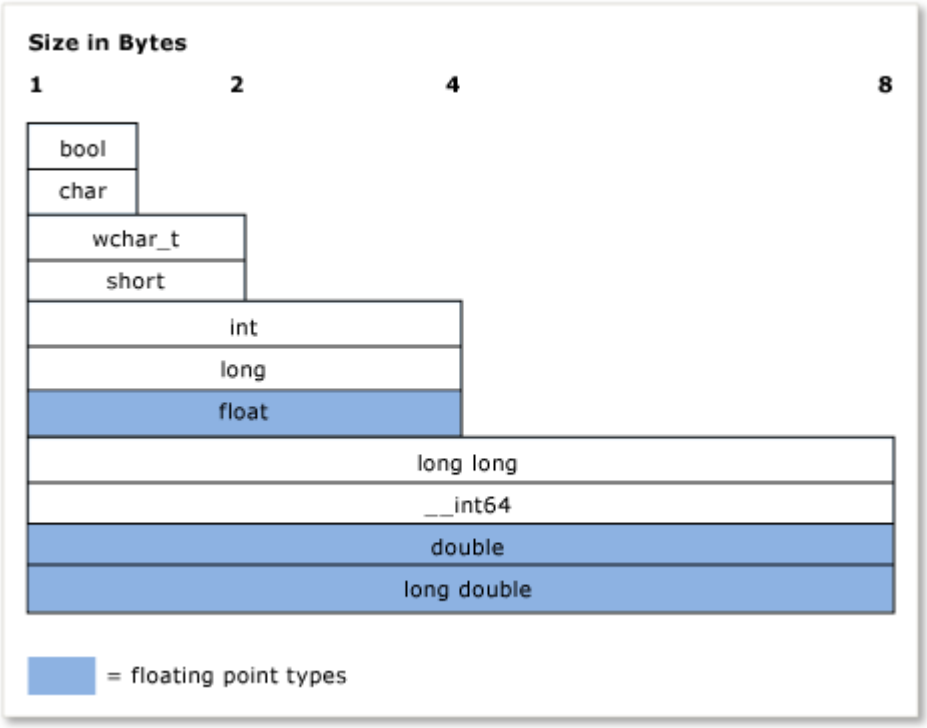
```
int result = 0;              // Declare and initialize an integer.
double coefficient = 10.8;   // Declare and initialize a floating
                             // point value.
auto name = "Lady G.";       // Declare a variable and let compiler
                             // deduce the type.
auto address;                // error. Compiler cannot deduce a type
                             // without an intializing value.
age = 12;                    // error. Variable declaration must
                             // specify a type or use auto!
result = "Kenny G.";         // error. Can't assign text to an int.
string result = "zero";      // error. Can't redefine a variable with
                             // new type.
int maxValue;                // Not recommended! maxValue contains
                             // garbage bits until it is initialized.
```

Fundamental (built-in) types

Unlike some languages, C++ has no universal base type from which all other types are derived. The Visual C++ implementation of the language includes many *fundamental types*, also known as *built-in types*. This includes numeric types such as `int`, `double`, `long`, `bool`, plus the `char` and `wchar_t` types for ASCII and UNICODE characters, respectively. Most fundamental types (except `bool`, `double`, `wchar_t` and related types) all have unsigned versions, which modify the range of values that the variable can store. For example, an `int`, which stores a 32-bit signed integer, can represent a value from -2,147,483,648 to 2,147,483,647. An `unsigned int`, which is also stored as 32-bits, can store a value from 0 to 4,294,967,295. The total number of possible values in each case is the same; only the range is different.

The fundamental types are recognized by the compiler, which has built-in rules that govern what operations you can perform on them, and how they can be converted to other fundamental types. For a complete list of built-in types and their size and numeric limits, see Fundamental Types.

The following illustration shows the relative sizes of the built-in types:



The following table lists the most frequently used fundamental types:

Type	Size	Comment
int	4 bytes	The default choice for integral values.
double	8 bytes	The default choice for floating point values.
bool	1 byte	Represents values that can be either true or false.
char	1 byte	Use for ASCII characters in older C-style strings or std::string objects that will never have to be converted to UNICODE.
wchar_t	2 bytes	Represents "wide" character values that may be encoded in UNICODE format (UTF-16 on Windows, other operating systems may differ). This is the character type that is used in strings of type `std::wstring`.
unsigned char	1 byte	C++ has no built-in `byte` type. Use unsigned char to represent a byte value.
unsigned int	4 bytes	Default choice for bit flags.
long long	8 bytes	Represents very large integer values.

The void type

The `void` type is a special type; you cannot declare a variable of type `void`, but you can declare a variable of type `void *` (pointer to `void`), which is sometimes necessary when allocating raw (un-typed) memory. However, pointers to `void` are not type-safe and generally their use is strongly discouraged in modern C++. In a function declaration, a `void` return value means that the function does not return a value; this is a common and acceptable use of `void`. While the C language

required functions that have zero parameters to declare `void` in the parameter list, for example, `fou(void)`, this practice is discouraged in modern C++ and should be declared `fou()`. For more information, see Type Conversions and Type Safety.

const type qualifier

Any built-in or user-defined type may be qualified by the const keyword. Additionally, member functions may be `const`-qualified and even `const`-overloaded. The value of a `const` type cannot be modified after it is initialized.

```
const double PI = 3.1415;
PI = .75 //Error. Cannot modify const variable.
```

The `const` qualifier is used extensively in function and variable declarations and "const correctness" is an important concept in C++; essentially it means to use `const` to guarantee, at compile time, that values are not modified unintentionally. For more information, see const.

A `const` type is distinct from its non-const version; for example, `const int` is a distinct type from `int`. You can use the C++ `const_cast` operator on those rare occasions when you must remove *const-ness* from a variable. For more information, see Type Conversions and Type Safety.

String types

Strictly speaking, the C++ language has no built-in “string” type; `char` and `wchar_t` store single characters – you must declare an array of these types to approximate a string, adding a terminating null value (for example, ASCII `'\0'`) to the array element one past the last valid character (also called a “C-style string”). C-style strings required much more code to be written or the use of external string utility library functions. But in modern C++, we have the Standard Library types `std::string` (for 8-bit `char`-type character strings) or `std::wstring` (for 16-bit `wchar_t`-type character strings). These STL containers can be thought of as native string types because they are part of the standard libraries that are included in any compliant C++ build environment. Simply use the `#include <string>` directive to make these types available in your program. (If you are using MFC or ATL, the CString class is also available, but is not part of the C++ standard.) The use of null-terminated character arrays (the C-style strings previously mentioned) is strongly discouraged in modern C++.

User-defined types

When you define a `class`, `struct`, `union`, or `enum`, that construct is used in the rest of your code as if it were a fundamental type. It has a known size in memory, and certain rules about how it can be used apply to it for compile-time checking and, at runtime, for the life of your program. The primary differences between the fundamental built-in types and user-defined types are as follows:

- The compiler has no built-in knowledge of a user-defined type. It “learns” of the type when it first encounters the definition during the compilation process.
- You specify what operations can be performed on your type, and how it can be converted to other types, by defining (through overloading) the appropriate operators, either as class members or non-member functions. For more information, see Overloading (C++).

- They do not have to be statically typed (the rule that an object's type never changes). Through the mechanisms of *inheritance* and *polymorphism*, a variable declared as a user-defined type of class (referred to as an object instance of a class) might have a different type at run-time than at compile time. For more information, see Inheritance.

Pointer types

Dating back to the earliest versions of the C language, C++ continues to let you declare a variable of a pointer type by using the special declarator * (asterisk). A pointer type stores the address of the location in memory where the actual data value is stored. In modern C++, these are referred to as *raw pointers*, and are accessed in your code through special operators * (asterisk) or -> (dash with greater-than). This is called *dereferencing*, and which one that you use depends on whether you are dereferencing a pointer to a scalar or a pointer to a member in an object. Working with pointer types has long been one of the most challenging and confusing aspects of C and C++ program development. This section outlines some facts and practices to help use raw pointers if you want to, but in modern C++ it's no longer required (or recommended) to use raw pointers for object ownership at all, due to the evolution of the smart pointer (discussed more at the end of this section). It is still useful and safe to use raw pointers for observing objects, but if you must use them for object ownership, you should do so with caution and very careful consideration of how the objects owned by them are created and destroyed.

The first thing that you should know is declaring a raw pointer variable will allocate only the memory that is required to store an address of the memory location that the pointer will be referring to when it is dereferenced. Allocation of the memory for the data value itself (also called *backing store*) is not yet allocated. In other words, by declaring a raw pointer variable, you are creating a memory address variable, not an actual data variable. Dereferencing a pointer variable before making sure that it contains a valid address to a backing store will cause undefined behavior (usually a fatal error) in your program. The following example demonstrates this kind of error:

```
int* pNumber;       // Declare a pointer-to-int variable.
*pNumber = 10;      // error. Although this may compile, it is
                    // a serious error. We are dereferencing an
                    // uninitialized pointer variable with no
                    // allocated memory to point to.
```

The example dereferences a pointer type without having any memory allocated to store the actual integer data or a valid memory address assigned to it. The following code corrects these errors:

```
    int number = 10;          // Declare and initialize a local integer
                              // variable for data backing store.
    int* pNumber = &number;   // Declare and initialize a local integer
                              // pointer variable to a valid memory
                              // address to that backing store.
...
    *pNumber = 41;            // Dereference and store a new value in
                              // the memory pointed to by
                              // pNumber, the integer variable called
                              // "number". Note "number" was changed, not
                              // "pNumber".
```

The corrected code example uses local stack memory to create the backing store that `pNumber` points to. We use a fundamental type for simplicity. In practice, the backing store for pointers are most often user-defined types that are dynamically-allocated in an area of memory called the *heap* (or "free store") by using a `new` keyword expression (in C-style programming, the older `malloc()` C runtime library function was used). Once allocated, these "variables" are usually referred to as "objects", especially if they are based on a class definition. Memory that is allocated with `new` must be deleted by a corresponding `delete` statement (or, if you used the `malloc()` function to allocate it, the C runtime function `free()`).

However, it is easy to forget to delete a dynamically-allocated object- especially in complex code, which causes a resource bug called a *memory leak*. For this reason, the use of raw pointers is strongly discouraged in modern C++. It is almost always better to wrap a raw pointer in a smart pointer, which will automatically release the memory when its destructor is invoked (when the code goes out of scope for the smart pointer); by using smart pointers you virtually eliminate a whole class of bugs in your C++ programs. In the following example, assume `MyClass` is a user-defined type that has a public method `DoSomeWork();`

```
void someFunction() {
    unique_ptr<MyClass> pMc(new MyClass);
    pMc->DoSomeWork();
}
  // No memory leak. Out-of-scope automatically calls the destructor
  // for the unique_ptr, freeing the resource.
```

For more information about smart pointers, see Smart Pointers.

For more information about pointer conversions, see Type Conversions and Type Safety.

For more information about pointers in general, see Pointers.

Windows data types

In classic Win32 programming for C and C++, most functions use Windows-specific typedefs and #define macros (defined in `windef.h`) to specify the types of parameters and return values. These "Windows data types" are mostly just special names (aliases) given to C/C++ built-in types. For a complete list of these typedefs and preprocessor definitions, see Windows Data Types. Some of these typedefs, such as HRESULT and LCID, are useful and descriptive. Others, such as INT, have no special meaning and are just aliases for fundamental C++ types. Other Windows data types have names that are retained from the days of C programming and 16-bit processors, and have no purpose or meaning on modern hardware or operating systems. There are also special data types associated with the Windows Runtime Library, listed as Windows Runtime base data types. In modern C++, the general guideline is to prefer the C++ fundamental types unless the Windows type communicates some additional meaning about how the value is to be interpreted.

More Information

For more information about the C++ type system, see the following topics.

Value Types	Describes *value types* along with issues relating to their use.
Type Conversions and Type Safety	Describes common type conversion issues and shows how to avoid them.

See Also

Welcome Back to C++
C++ Language Reference
C++ Standard Library