**Language Standards**

Object-oriented languages have evolved since the concept of objects was formally incorporated into the Simula language by Ole-Johan Dahl and Kristen Nygaard in the 1960s.  International standards have documented their evolution.  The most recent standard for the C++ language is formally known as ISO/IEC 14882:2014 and extends over 1300 pages.  ISO stands for the International Organization for Standardization.  IEC stands for the International Electrotechnical Commission.

This chapter reviews the milestones in the evolution of C++, highlights some of the features introduced during its evolution with respect to the original version of the language, and briefly discusses a few of the topics that have evolved with the C++ standards.

Milestones

C++ was originally designed as a synthesis of C and object-orientation Simula-style.  C had and still has no object-oriented capabilities.  Simula introduced the terms class, object, inheritance, virtual methods and subclasses (derived classes) formally to the programming community.

[Bjarne Stroustrup](http://www.research.att.com/~bs/homepage.html) created C++ at Bell Labs (AT&T Research Labs) by augmenting C with the object-oriented features of Simula.  He released C++ officially in October 1985.  His web site includes a quite useful and up-to-date [glossary of technical terms](http://www.research.att.com/~bs/glossary.html).

The ISO/IEC Standards

At the time of printing, three standard definitions have been approved by the international programming community.

* C++98
* C++11
* C++14

C++98

The first official standard that defined the C++ language is formally known as ISO/IEC 14882:1998 and less formally as C++98.  The international programming community ratified this definition in 1998 and published it in a document that contains about 800 pages.  The definition is based in part on the ISO/IEC 9899:1990 standard for the C language (informally known as C89).

C++98 augmented pre-standard C++ with

"additional data types, classes, templates, exceptions, namespaces, inline functions, operator overloading, function name overloading, references, freestore management operators, and additional library facilities."

The library facilities included a newly re-written iostream library and the string class.

C++11

The second official standard that re-defined the language is formally known as ISO/IEC 14882:2011 and less formally as C++11.  The international programming community ratified this definition on August 12 2011.  The definition is based on C++98 and C99 and includes several major additions to the core language as well as several major extensions of the standard library.

The objectives of the C++11 committee had included:

* making C++ easier to teach and to learn through increased uniformity
* making C++ better for systems programming and library construction
* improving the type safety of the language

The features that C++11 added to C++98 included (amongst others):

* the nullptr keyword replacing the NULL macro
* the auto keyword inferring the type of a left operand implicitly from the type of the right operand in an assignment expression
* inherited constructors
* features covered in the next volume of this series of notes
  + move constructors and assignment operators
  + lambda expressions (anonymous functions)
  + library support for multi-threading classes
  + range based for loops
  + strongly typed enumerations
  + uniformity amongst initializers
  + initializers for class members

C++14

The third official standard that re-defined the language is formally known as ISO/IEC 14882:2014 and less formally as C++14.  The international programming community ratified this definition on August 18 2014.  The definition expanded the application of the auto keyword to return types, the application of templates to variables.  The changes made the language safer and more convenient.

These notes align with this standard.

C++17

The ISO/IEC 14882 standards committee is working on the next iteration scheduled for ratification in July 2017.  The committee intended this revision to be a major amendment to the C++14 standard, but several new features did not make the cut.

Compiler Support Status

A language standard is a specification for compiler writers.  Different writers introduce different features adopted in a standard at different times.  The support status for the features approved in C++11 and C++14 is tabulated at <http://en.cppreference.com/w/cpp/compiler_support>.  Links to the individual compiler web sites are included there.

Some Features that have Changed

Facilities that C++98 and C++11 introduced included inline functions, member function deletion, constrained casting and changes to freestore management.

Inline Functions

*Inlining* is a technique for improving a function's execution time by replacing the function call with the function logic proper and thereby removing the overhead associated with parameter passing.  The primary cost of inlining is an increase in the size of the executable code.

An inline request directs the compiler to insert the body of the function at every call to the function, if possible.  The compiler, instead of storing the function's definition once in its own dedicated region of memory and transferring control to that region for each call, inserts a copy of the body at each and every call.  Inlining is particularly useful with member functions that contain small blocks of code.  Member functions that do not contain iterations are candidates for inlining.

The compiler determines whether or not to implement an inline request.  If the function contains too many statements or an iteration, the compiler ignores the request and calls the function in the usual way.

Example

To inline a member function, we embed its definition within the class definition as shown on the left or alternatively add the keyword inline to the definition as shown on the right.

For example,

|  |  |
| --- | --- |
| // Inline Functions - Embedded  // inline\_1.h  const int NG = 20;  struct Student {  private:  int no;  float grade[NG];  int ng;  public:  void set(int n, const char\* g);  const float\* getGrades() const {  return grade;  }  }; | // Inline Functions - Separate  // inline\_2.h  const int NG = 20;  struct Student {  private:  int no;  float grade[NG];  int ng;  public:  void set(int n, const char\* g);  const float\* getGrades() const;  };  inline const float\* Student::getGrades()  const {  return grade;  } |

Note that we place the implementation of an inline function in the header file that includes the class definition.

Function Deletion

C++11 introduced use of the keyword delete to inform the compiler that we do not want any default definition of a member function created.

The Legacy Way

Prior to C++11, one way to prohibit a client from copying or copy assigning an instance of a class was to declare both the copy constructor and copy assignment operator as private members:

|  |
| --- |
| class Student {  int no;  float\* grade;  int ng;  Student(const Student& source);  Student& operator=(const Student& source);  public:  Student();  Student(int, const float\*);  ~Student();  void display() const;  }; |

Since these special member functions were defined as private members, no definition was practically necessary.

C-Style Casts

C++ inherited its original casting facilities from C directly.  The constrained casting syntax is more discriminating than the inherited syntax.  The standards support the inherited syntax for legacy reasons.  The availability of these older features allows programmers to bypass the type system and directly weaken a compiler's ability to identify type errors.

For example, consider code that converts an int to a pointer to an int.  Such code is most probably a typing mistake, C and hence C++ allow this code to slip through the type checking system:

|  |
| --- |
| int x = 2;  int\* p;  p = (int\*)(x); // MOST PROBABLY A TYPING ERROR (& missing)! |

Nevertheless, in applications built from many thousands of lines of code, we expect the compiler's type-checking system to flag such code.  Errors that result from such casts are very difficult to find if they are embedded within many thousands of lines of code.

C++ supports old-style casting in two distinct forms - plain C-style casts and C++-function-style casts:

(*Type*) *identifier* and *Type* (*identfier*)

These forms are interchangeable for fundamental types, but not pointer types.  For conversions to pointer types, only the C-style cast is available.

C-Style Casts

To cast a value from one type to another using a C-style cast, we simply preface the identifier with the name of the target type enclosed in parentheses:

|  |
| --- |
| // C-Style Casting  // c\_cast.cpp  #include <iostream>  int main() {  double hours;  int minutes;  std::cout << "Enter minutes : ";  std::cin >> minutes;  hours = (double) minutes / 60; // C-Style Cast  std::cout << "In hours, this is " << hours;  } |

Function-Style Casts

To cast a value from one type to another using a function-style cast, we enclose in parentheses the variable or object to be cast to the target type:

|  |
| --- |
| // Function Style Casting  // functionStyleCast.cpp  #include <iostream>  int main() {  double hours;  int minutes;  std::cout << "Enter minutes : ";  std::cin >> minutes;  hours = double(minutes) / 60; // Function-Style Cast  std::cout << "In hours, this is " << hours;  } |

Comparison

The C-style casts (for example, (int)x) apply without regard to the nature of the conversion.  Such syntax does not convey the programmer's intent.

A C-style cast can mean any of the following:

* static\_cast
* const\_cast
* static\_cast + const\_cast
* reinterpret\_cast
* reinterpret\_cast + const\_cast

The constrained casts distinguish the different categories and thereby improve the degree of type checking available from the compiler.

For example, it is always safer type-wise to code a static\_cast rather than a C-style cast.

Freestore Management

C++98 introduced exception handling for dynamic memory allocation.  By default, the new operator throws an exception if the operator encounters an error.  The topic of exception handling is covered in the next volume of this series of notes.

The Legacy Way

Prior to C++98, the new operator returned the null address if it encountered an error (for example, insufficient memory).

The following legacy code checks for such an error:

|  |
| --- |
| // Prior to C++98  #include <iostream.h>  int main() {  char\* p;  int i = 0;  do {  p = new char[100001];  i++;  } while (p != NULL);  cout << "Out of space after " << i << " attempts!\n";  } |

One Alternative

Since C++98, we can instruct the new operator to return the null address by passing the nothrow argument to the operator.  nothrow is defined in the new header file:

|  |
| --- |
| // After C++98 - Null Address Alternative  #include <new>  #include <iostream>  int main() {  char\* p;  int i = 0;  do {  p = new (std::nothrow) char[100001];  i++;  } while (p != NULL);  std::cout << "Out of space after " << i << " attempts!\n";  } |

With C++11, we can improve type safety by replacing the macro NULL with the nullptr keyword:

|  |
| --- |
| // After C++11 - Null Address Alternative  #include <new>  #include <iostream>  int main( ) {  char\* p;  int i = 0;  do {  p = new (std::nothrow) char[100001];  i++;  } while (p != nullptr);  std::cout << "Out of space after " << i << " attempts!\n";  } |

A Technical Note on Inclusion Polymorphism

Dynamic Dispatch or Late Binding

The terms dynamic dispatch and late binding arise in descriptions of inclusion polymorphism.  These terms have similar definitions and are sometimes used interchangeably.  Technically, dynamic dispatch is the more precise term in regard to C++.

*Dynamic dispatch* is the process of selecting which implementation of a member function in a class hierarchy to call on a polymorphic object.  The name of the operation may be bound to a polymorphic operation at compile time and the implementation identified at run time.  The object's dynamic type determines which implementation to call.

*Late binding* associates a method's name with an object based on its dynamic type.  The name of the operation is bound to a polymorphic operation at run time when the implementation is identified.  Late binding implies dynamic dispatch.

C++ uses early binding and static or dynamic dispatch.  Static dispatch is the default.  The virtual keyword implements dynamic dispatch.

Virtual Table

Most C++ compilers implement dynamic dispatch by adding an instance pointer to the object's data members.  This pointer redirects to a table that associates the member function implementations with object types.  The run-time code uses this table to select the implementation corresponding to the object's dynamic type.  This table is called the *virtual table* for the class.

The compiler creates the virtual table at compile time.  Introducing a virtual table (by inserting the keyword virtual) results in the equivalent of a single indirection every time a client calls a virtual member function on an instance of its class; that is, it does not introduce a significant overhead.  Since the compiler constructs the table at compile time, it is unmodifiable at run-time and we cannot add a new member function to the class at that time.