Differences Between ANSI C, C++ and ANSI C++

A.1 INTRODUCTION

C++ is a superset of ANSI C. It has many additional features, which can be grouped into two categories, namely,

1. features that are improvements over ANSI C to enhance programming style in a non-object-oriented environment, and
2. features that are added to aid object-oriented programming.

Although a majority of the programs written in ANSI C will run under C++ compiler, a few may not. They may require minor modifications before they are compatible to the C++ environment. This appendix summarizes instances where an ANSI C program is not a C++ program, and highlights the features added to ANSI C to make it an object-oriented programming language.

A.2 WHERE ANSI C IS NOT C++

Keywords

In order to add features to C, a number of new keywords have been added to ANSI C. These are:

C++ additions

asm	**catch**	**class**	**delete**
friend	**inline**	**new**	**operator**

private	**protected**	**public**	**template**
this	**throw**	**try**	**virtual**

ANSI C++ additions

bool	**export**	**reinterpret_cast**	**typename**
const_cast	**false**	**static_cast**	**using**
dynamic_cast	**mutable**	**true**	**wchar_t**
explicit	**namespace**	**typeid**	

ANSI C programs using any of these keywords as identifiers are not C++ programs.

Length of Identifiers

C++ does not put any limit on the length of identifiers. Some C implementations may impose restrictions on the length.

Declarations

In ANSI C, a global variable can be declared several times without using the **extern** specifier. C++ does not permit this. It must be declared only once without **extern** and all the remaining must use the **extern** specifier.

Global const

In ANSI C, a global **const** has an external linkage. But it is not true in C++ where all **const** values are **static** by default and thus become local.

Character Constants

In ANSI C, the character constants are of type **int**. C++ treats the character constants as type **char**. This is necessary because the compiler needs to distinguish between the two overloaded functions based on their arguments. For example, the declarations such as

```
void display(int);
void display(char);
```

are valid in C++. When we call the function

```
display('A');
```

the compiler will call **display(char)** and not **display(int)**.

Enumerated Data Type

The **enum** data type differs slightly in C++. The **enum** tag name is considered a type name in C++. ANSI C defines the type **enum** to be **int**, whereas in C++, each enumerator is the type of its enumeration. This means that C++ does not allow for an **int** value to be automatically converted to an **enum** value. Example:

```
enum colour {yellow, green, blue};
colour paint;          // declare paint of type colour
paint = green;         // valid, green is of type colour
paint = 5;             // invalid, 5 is of type int
paint = colour(5);     // valid
```

Another difference is that ANSI C allows an **enum** to be defined within a structure, but the **enum** is globally visible. In C++, an **enum** defined within a structure is only visible inside the structure.

The C++ also supports the creation of anonymous enum's (i.e. enum's without tags) as shown below:

```
enum{yellow, green, blue};
```

Main Function

C++ forces the **main()** function to return a value of type **int**. Therefore, all **main()** functions should have an explicit **return(0)** statement.

Function Prototype

The supply of function prototypes is a must in C++. The function declaration must define not only the return type but also the types of parameters the function is going to use. This is not essential in ANSI C. Therefore, all the old C type functions will work under ANSI C but will not in C++.

Functions with No Arguments

Functions declared with an empty argument list are treated differently in C++ and ANSI C. For example, a function declaration such as

```
int test();
```

means 'unspecified number of arguments' in ANSI C and 'exactly zero arguments' in C++. Therefore, the function **test()** can be called with arguments in ANSI C. But it will be an error in C++. Function declarations with no arguments for use in both ANSI C and C++ programs should be declared explicitly as follows:

```
int test(void);
```

Function Header

K&R C defines a function header as follows:

```
double mul(m,a)
int m;
float a;
```

```
{
        .....
        .....
}
```

While this is acceptable in ANSI C as well as C++, this style is likely to be dropped in the future versions of C++. It is better to define the function header using a prototype-like format as shown below:

```
double mul (int m, float a)
{
        .....
        .....
}
```

void Data Type

The void specifier can be used to define a pointer to a generic item. There is a significant difference in how C++ and ANSI C handle the assignment of **void** pointers to other pointer types. In ANSI C, we can assign void pointer to other pointer types without using a cast.

Example:

```
void *a;
char *c;
c = a;          // assign void pointer to char pointer
```

This will not work in C++. A void pointer cannot be assigned directly to other type pointers in C++. This can be done using the cast operator as follows:

```
c = (char *) a;
```

Remember, in both the cases, a pointer of type void can be assigned any type. That is

```
a = c;
```

will work both in ANSI C and C++.

const Variables

const means different things in C++ and ANSI C. In ANSI C, it means that 'it cannot be changed' to any ordinary variable. In C++, it is intended to replace the use of **#define** to create the symbolic constants. The symbolic constants have no type information whereas **const** values in C++ can have types specified explicitly. For example, the statement

```
const int N = 100;
```

creates a constant **N** of type **int**.

This information can help the compiler detect errors such as calling a function with arguments of the wrong type.

The other main difference is that in C++, we can use a **const** in a constant expression. For example, we can use **const** value to declare the size of an array.

```
const size = 30;
char buffer[size];
```

This is illegal in C. Note that when we omit the type, C++ uses **int**. One another difference is that C++ requires the **const** variables to be initialized at the time of declaration. ANSI C does not. That is,

```
const max;
```

is legal in ANSI C, but not in C++. This must be written as

```
const max = 100;
```

Character Array Initialization

When initializing character arrays in ANSI C, we can declare the array size as equivalent to the number of characters in the string. Example:

```
char name[4] = "GOOD";
```

That is, we need not provide any extra space for the terminating character \0. The above initialization is not legal in C++. The size of the array must include an extra space for holding the null character \0.

goto Jump

ANSI C allows a **goto** to jump into a block of code, past a variable declaration and initialization. Example:

```
.....
goto label1;
.....
{
    int x = 1;
    .....
    .....
    label1;
}
.....
.....
```

C++ does not permit a **goto** to skip an initialization as shown above.

A.3 C++ EXTENSIONS TO ANSI C

C++ is an attempt to improve upon C by expanding its features to support new software development concepts such as object-oriented programming. This section summarizes the new features that are added to the ANSI C.

Comments

C++ supports one more type of comment in addition to the existing comment style /*.....*/. This is useful for one-line comments as shown below:

```
// This is a C++ one-line comment
```

In general, the /*.....*/ style is used for a group of lines and the //.... style is used for the single line comments.

Casts

C++ introduces a new type of cast syntax as shown below:

```
m = long(n);     // new C++ cast
```

C++ also supports C-type cast:

```
m = (long) n;    // C-type cast
```

Declarations

Unlike ANSI C which requires all the declarations to be made at the beginning of the scope, C++ permits us to make declarations at any point in the program. This enables us to place the declarations closer to the point of their use.

Scope Resolution

A new operator :: known as the scope resolution operator has been introduced to access an item that is outside the current scope. This operator is also used for distinguishing class members and defining class methods.

New Keywords

In order to add new features, it was necessary to create a number of new keywords. They are listed in Section A.2 in this Appendix.

Struct and Union Tags

In C++, both structs and unions are treated as types of classes and their tag names as type names. For example,

```
struct item{int cost; float weight};
item a;
```

are valid statements in C++. However, the older syntax is also acceptable. The structs and unions can contain functions as members in addition to data members.

Anonymous Unions

In C++, we can declare unions without tag names. For example,

```
union
{
      int height;
      float weight;
}
```

is a legal declaration in C++. The members can be directly accessed by the name.

```
height = 175;
weight = 70;
```

Free Store Operators

C++ defines two new operators **new** and **delete** to manage the dynamic memory allocation function. They can be used in places of **malloc()** and **free()** used in ANSI C. Example:

```
void fun(void)
{
      float  *p;
      p = new float;
      *p = 25.79;
      delete p;
}
```

References

C++ allows the use of a reference, which is simply another name of the variable that is being initialized.

```
int m = 100;
int &x = m;
```

x is an alternative name for **m**. If we add, say 10, to **x**:

```
x =  x + 10;
```

we have, in effect, added 10 to **m**.

Inline Functions

To eliminate the cost of calls to small functions, C++ supports a concept known as inline functions. These functions are specified with keyword **inline** as shown below:

```
inline int mul(int a, int b)
{
        return (a * b);
}
```

An *inline* function is inserted wherever a call to that function appears. This is called *inline expansion*. All inline functions must be defined before they are called.

Default Function Arguments

In C++, we can force the arguments to default to predefined values. These values are specified in the function prototype as shown below:

```
void func1(int a, int b=2, int c=3);
```

Typical function calls could be:

```
func1(5,10);
func1(10);
func1(5,10,15);
```

Function Overloading

Function names can be overloaded in C++. That is, we can assign the same name to two or more distinct functions. Example:

```
int mul(int a, int b);          // prototype
float mul(float x, float y);    // prototype
```

The function name **mul()** is overloaded, meaning that it has more than one implementation. The execution of the correct implementation is decided by the compiler by matching the type of the arguments in the function call with the types in the function prototypes.

Operator Overloading

C++ allows overloading of most operators. This feature allows us to change the meaning of the operators. For example, we can add two user-defined data types using the operator +. The concept of operator overloading is extensively used in manipulating the class type objects. Example:

```
Object1 = Object1 + Object2;
```

C++ works by creating functions that define the actions performed on non-intrinsic types by an operator. *An operator function* is declared using the **operator** keyword and the operator symbol.

Classes

C++ extends the syntax of struct to allow the inclusion of functions in structures. The new syntax is termed as *class* in C++. Classes offer a method of grouping together data and functions that can operate on this data. They also provide a mechanism for restricted access to specific data.

It is also possible to derive a class from one or more base classes. A derived class inherits all the members of its base class. Access privileges of the inherited members can be changed by the derived class through access specifiers. Derived classes provide the capability of building a hierarchical structure of objects.

The classes provide the programmer with a tool for creating new data types that can be used as conveniently as the built-in types.

Pointers

Pointers are declared and referenced similar to C. However, C++ adds the concepts of constant pointer and pointer to constant:

```
char * const ptr1;      // constant pointer
```

We cannot modify the address that **ptr1** is initialized to:

```
char const * ptr2;      // pointer to a constant
```

The contents of what **ptr2** is pointing to cannot be changed through indirection.

Templates

Templates, a feature added to C++, enable us to define generic classes and functions that can be used to create a family of classes and functions with different data types.

Exception Handling

C++ adds a mechanism to locate and manage certain disastrous error conditions known as exceptions during the execution of a program.

Input/Output in C++

All I/O functions available in C are supported by C++ as well. However, in addition to what C provides, C++ adds a few unique features of its own. It supports several functions that allow reading and writing of data in both formatted and unformatted forms.

A.4 ANSI C++ ADDITIONS TO C++

ANSI C++ has added several new features to the original C++ specifications. Important features added are:

Data types

- bool
- wchar_t

Operators

- const_cast
- static_cast
- reinterpret_cast
- typeid

Class member qualifiers

- Explicit constructor
- Mutable members
- Namespace scope
- Operator keywords
- New style headers
- New keywords

These features are discussed in detail in Chapter 16.

A.5 STANDARD TEMPLATE LIBRARY

A large number of generic classes and functions have been developed that could be used as a standard approach for storing and processing of data. These classes and functions, collectively known as the Standard Template Library, have now become a part of ANSI C++ class library. The important features of the Standard Template Library are presented in Chapter 14.