OBJECT-ORIENTED PROGRAMMING

Lesson 2

Tokens, Expressions and Control Structures

Tokens , Expressions and Control Structures

1. Introduction
2. Tokens
3. Keywords
4. Identifiers and Constants
5. Basic Data Types
6. User-Defined Data Types
7. Derived Data Types
8. Symbolic Constants
9. Type Compatibility
10. Declaration of Variables
11. Dynamic Initializations of Variables
12. Reference Variables

Tokens , Expressions and Control Structures

13. Operators in C++
14. Scope Resolution Operator
15. Member Dereferencing Operators
16. Memory Management Operators
17. Manipulators
18. Type Cast Operator
19. Expressions and Their Types
20. Special Assignment Expressions
21. Implicit Conversions
22. Operator Overloading
23. Operator Precedence
24. Control Structures

Key Concepts

- Tokens
- Keywords
- Identifiers
- Data Types
- User-defined types
- Derived types
- Symbolic constants
- Declaration of variables
- Initialization
- Reference variables
- Type compatibility
- Scope resolution
- Dereferencing
- Memory management
- Formatting the output
- Type casting
- Constructing expressions
- Special assignment expressions
- Implicit conversion
- Operator overloading
- Control structures

Introduction

- C++ is a superset of C and therefore most constructs of C are legal in C++ with their meaning unchanged.
- However, there are some exceptions and additions.

Tokens

- Tokens : The smallest individual units in a program.
- C++ has the following tokens:
  - Keywords
  - Identifiers
  - Constants
  - Strings
  - Operators

Keywords

- The keyword implement specific C++ language features.
- They are explicitly *reserved identifiers* and cannot be used as names for the program variables or other user-defined program element.

C++ keywords

asm	***double***	new	***switch***
auto	***else***	operator	template
break	***enum***	private	this
case	***extern***	protected	throw
catch	***float***	public	try
char	***for***	***register***	***typeof***
class	friend	***return***	***union***
const	***goto***	***short***	***unsigned***
continue	***if***	***signed***	virtual
default	inline	***sizeof***	***void***
delete	***int***	***static***	***volatile***
do	***long***	***struct***	***while***

Note: The ANSI C keywords are shown in bold face.

Keywords

- Additional keywords have been added to the ANSI C keywords in order to enhance its features and make it an object-oriented language.

Keywords (Added by ANSI C++)

- ANSI C++ standards committee has added some more keywords to make the language more versatile.

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

Identifiers and Constants

- ***Identifiers***
  - Refer to the name of variables, functions, arrays, classes, etc. created by the programmer.
  - Rules for naming identifiers
    - Only alphabetic characters, digits and underscores are permitted.
    - The name cannot start with a digit.
    - Uppercase and lowercase letters are distinct.
    - A declared keyword cannot be used as a variable name.

Identifiers and Constants

- *Identifiers*
  - A major difference between C and C++ is the limit on the length of a name.
  - ANSI C recognizes only the first 32 characters in a name.
  - ANSI C++ places no limit on its length.

Identifiers and Constants

- ***Constants***
  - Refer to fixed values that do not change during the execution of a program.

Identifiers and Constants

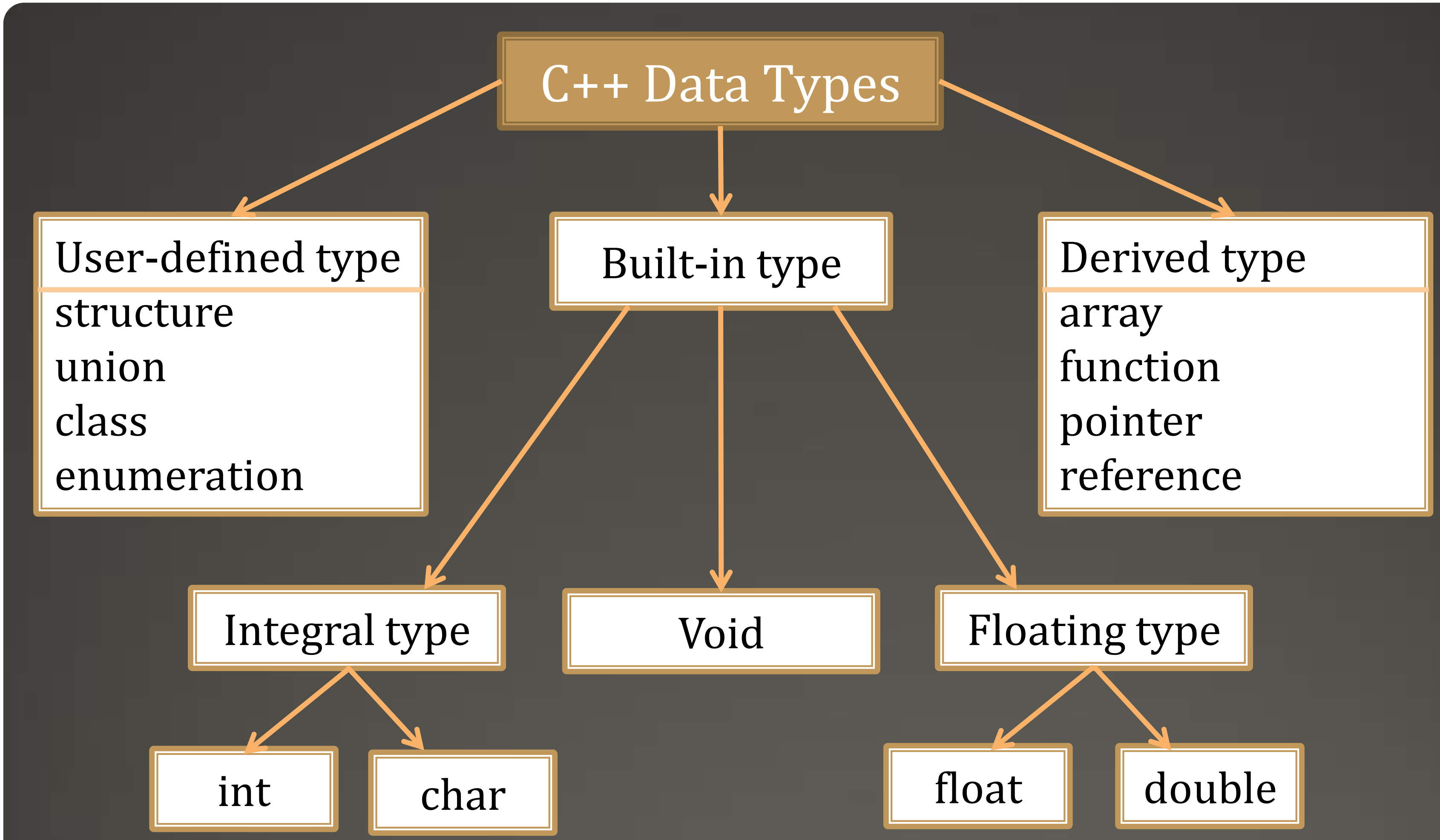
- ***Constants***
  - Like C, C++ supports several kinds of *literal constants* : integers, characters, floating point numbers and strings.
  - Examples:

```
123          //decimal integer
12.34        //floating point number
037          //octal integer
0X2          //hexadecimal integer
“C++”        //string constant
‘A’          //character constant
L’ab’        //wide-character constant
```

Identifiers and Constants

- ***Constants***
  - The *wchar_t* type is a wide-character literal introduced by ANSI C++ and is intended for character sets that cannot fit a character into a single byte.
  - Wide-character literals begin with the letter **L**.

Basic Data Types



C++ Data Types

User-defined type
structure
union
class
enumeration

Built-in type

Derived type
array
function
pointer
reference

Integral type

Void

Floating type

int

char

float

double

Hierarchy of C++ data types

Basic Data Types

- Both C and C++ compilers support all the *built-in* (*basic* or *fundamental*) data types.
- With the exception of *void*, the *basic data types* may have several *modifiers* preceding them to share the needs of various situations.
- The *modifiers signed*, *unsigned*, *long* and *short* may be applied to character and integer basic data types.
- The modifier *long* may also be applied to *double*.
- Data type representation is machine specific in C++.

Type	Bytes	Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-31768 to 32767
short int	2	-31768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
signed long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

Size and range of C++ basic data types for a 16-bit word machine

Basic Data Types

- *Void*
  - The type *void* was introduced in ANSI C.
  - Two normal uses' Void are

    1. To specify the return type of a function when it is not returning any value.
    2. To indicate an empty argument list to a function.

       Example:

```
void funct1(void);
```

Basic Data Types

- *Void*
  - Another interesting use of void is the declaration of *generic pointers*.
    - Example:

      ```
      void  *gp;             //gp becomes generic pointer
      ```

    - A generic pointer can be assigned a pointer value for any basic data type, but it may not be dereferenced.
    - For example

      ```
      int  *ip;              //int pointer
      gp = ip;               //assign int pointer to void pointer
      ```

    - are valid statements.
    - The statement `*ip = *gp;` is illegal.
    - It would not make sense to dereference a pointer to a *void* value.

Basic Data Types

- *Void*
  - Assigning any pointer type to a *void* pointer without using a cast is allowed in both C++ and ANSI C.
  - In ANSI C, we can also assign a *void* pointer to a non-void pointer without using a cast to non-void point type.
  - This is not allowed in C++.

    For example are all valid statements in ANSI C but not in C++.

    ```
    void  *ptr1;
    char  *ptr2;
    ptr2 = ptr1;
    ```

  - We need to use a cast operator

    ```
    ptr2 = (char  *) ptr1;
    ```

User-Defined Data Types

- Structures and Classes
  - Some user-defined data types such as *struct* and *union* in C are legal in C++.
  - And some more features have been added to make them suitable for object-oriented programming.
  - *Class*
    - Another user-defined data type that C++ permits us to define which can be used, just like any other basic data type, to declare variables.
  - *Object*
    - The class variable which is the central focus of object-oriented programming.

User-Defined Data Types

- Enumerated Data Type
  - Provide a way for attaching names to numbers, thereby increasing comprehensibility of the code.
  - The ***enum*** keyword (from C) automatically enumerates a list of words by assigning them values 0,1,2, and so on.
  - The syntax of an ***enum*** statement is similar to that of the struct statement.

```
enum  shape{circle, square, triangle};
enum  colour{red, blue, green, yellow};
enum  position{off, on};
```

User-Defined Data Types

- Enumerated Data Type
  - ANSI C defines the types of enums to be ints.
  - In C++, each enumerated data type retains its own separate type.
  - C++ does not permit an int value to be automatically converted to an enum value.
  - Examples

```
colour background = blue;            //allowed
colour background = 7;               //Error in C++
colour background = (colour) 7;   //OK
```

  - However, an enumerated value can be used in place of an int value.

```
int c = red;          //valid, colour type promoted to int
```

User-Defined Data Types

- Enumerated Data Type
  - By default, the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second, and so on.
  - We can over-ride the default by explicitly assigning integer values to the enumerators.
  - For example

```
enum colour {red, blue=4, green=8};
enum colour {red=5, blue, green}
```

    - In the first case, red is 0 by default.
    - In the second case, blue is 6 and green is 7.
  - Note that the subsequent initialized enumerators are larger by one than their predecessors.

User-Defined Data Types

- Enumerated Data Type
  - C++ also permits the creation of anonymous enums (i.e. enums without tag names).
  - Example

    ```
    enum {off, on};
    ```

    - Here, off is 0 and on is 1.
  - These constants can be referenced in the same manner as regular constants.
  - Examples

    ```
    int switch_1 = off;
    int switch_2 = on;
    ```

User-Defined Data Types

- Enumerated Data Type
  - In practice, enumeration is used to define **symbolic constants** for a **switch** statement.
  - Example

```
enum shape {circle, rectangle, triangle};
int main ()
{
        cout <<  “Enter shape code : ”;
        int code;
        cin >> code;
        while (code >= circle && code <= triangle)
        {
                Switch (code)
                {
                        case circle:
                        ……
                        break;
                        case rectangle:
                        ……
                        break;
                        case triangle:
                        ……
                        break;
                }
                cout << “Enter shape code: ”;
                cin >> code;
        }
        cout << “BYE  \n”;
        return 0;
}
```

User-Defined Data Types

- Enumerated Data Type
  - ANSI C permits an enum to be defined within a structure or a class, but the enum is **globally visible**.
  - In C++, an enum defined within a class (or structure) is **local** to that class (or structure) only.

Derived Data Types

- Arrays
  - The application of arrays in C++ is similar to that in C.
  - The only exception is the way character arrays are initialized.
    - In ANSI C, the compiler will allow us to declare the array size as the exact length of the string constant.
    - For instance `char string[3] = “xyz”;`
    - It assumes that the programmer intends to leave out the null character \0 in the definition.
    - But in C++, the size should be one larger than the number of characters in the string.

```
char string[4] = “xyz”;       //O.K.  for C++
```

Derived Data Types

- Functions
  - Functions have undergone major changes in C++.
  - Many of these modifications and improvements were driven by the requirements of the **object-oriented** concept of C++.

Derived Data Types

- Pointers
  - Pointers are declared and initialized as in C.
  - Examples

```
int  *ip;                //int pointer
ip = &x;                 //address of x assigned to ip
*ip = 10;                //10 assigned to x through indirection
```

Derived Data Types

- Pointers
  - C++ adds the concept of **constant pointer** and **pointer to a constant**.

```
char  * const ptr1 = “GOOD”;                    //constant pointer
```

  - We cannot modify the address that ptr1 is initialized to.

```
int  const  * ptr2 =  &m;                       //pointer to a constant
```

  - ptr2 is declared as pointer to a constant.
  - It can point to any variable of correct type, but the contents of what it points to cannot be changed.

Derived Data Types

- Pointers
  - We can also declare both the pointer and the variable as constants in the following way:

```
const  char * const cp = “xyz”;                    //pointer to a constant
```

  - This statement declares cp as a constant pointer to the string which has been declared a constant.
    - In this case, neither the address assigned to the pointer cp nor the contents it points to can be changed.

Derived Data Types

- Pointers
  - Pointers are extensively used in C++ for memory management and achieving polymorphism.

Symbolic Constants

- Two ways of creating symbolic constants in C++:
  - Using the qualifier ***const***
  - Defining a set of integer constants using ***enum*** keyword
- In both C and C++, any value declared as ***const*** cannot be modified by the program in any way.

Symbolic Constants

- Differences in C and C++ in implementation
  - In C++, we can use **const** in a constant expression, such as

```
const  int  size = 10;
char  name[size];
```

    - This would be illegal in C.
    - **const** allows us to create typed constants instead of having to use **#define** to create constants that have no type information.

Symbolic Constants

- Differences in C and C++ in implementation
  - As with **long** and **short**, if we use the const modifier alone, it defaults to int.
    - For example `const size = 10` means `const  int  size = 10`
  - The **named constants** are just like variables except that their values cannot be changed.

Symbolic Constants

- Differences in C and C++ in implementation
  - C++ requires a const to be initialized.
  - ANSI C does not require an initializer;
    - If none is given, it initializes the const to 0.

Symbolic Constants

- Differences in C and C++ in implementation
  - The scoping of const values differs.
    - A const in C++ defaults to the internal linkage and therefore it is local to the file where it is declared.
    - In ANSI C, const values are global in nature.
      - They are visible outside the file in which they are declared.
      - However, they can be made local by declaring them as static.
    - To give a const value an external linkage so that it can be referenced from another file, we must explicitly define it as an **extern** in C++.
    - Example :

```
extern  const  total = 100;
```

Symbolic Constants

- Another method of naming integer constants is by enumeration as

```
enum {X, Y, Z};
```

  - This defines X, Y and Z as integer constants with values 0, 1 and 2 respectively.
  - This is equivalent to

```
const  X = 0;
const  Y = 1;
const  Z = 2;
```

  - We can also assign values to X, Y and Z explicitly.
    - Example

```
enum {X = 100, Y = 50, Z = 200};
```

    - Such values can be any integer values.

Type Compatibility

- C++ is very strict with regard to type compatibility as compared to C.
  - For instance, C++ defines **int**, **short int**, and **long int** as three different types.
  - They must be cast when their values are assigned to one another.
  - Similarly, **unsigned char**, **char**, and **signed char** are considered as different types, although each of these has a size of one byte.

Type Compatibility

- In C++, the types of values must be the same for complete compatibility, or else, a cast must be applied.
  - These restrictions in C++ are necessary in order to support function overloading where two functions with the same name are distinguished using the type of function arguments.

Type Compatibility

- Another notable difference is the way char constants are stored.
  - In C, they are store as **ints**.
  - And therefore `sizeof ('X')` is equivalent to `sizeof (int)` in C.
  - In C++, char is not promoted to the size of int.
  - And therefore `sizeof ('X')` equals `sizeof (char)`

Declaration of Variables

- In C, all variables must be declared before they are used in executable statements.
- This is true with C++ as well.
- A significant difference between C and C++ regarding the place of their declaration in the program.

Declaration of Variables

- In C
  - It is required that all the variables to be defined at the beginning of a scope.
- In C++
  - It allows the declaration of a variable anywhere in the scope.
  - This means that a variable can be declared right at the place of its first use.

Declaration of Variables

- Example 1

```
int main( )
{
        float  x;               //declaration
        float  sum = 0;

        for (int  i=1; i<5; i++)
        {
                cin >> x;
                sum = sum +x;
        }
        float  average;     //declaration
        average = sum/(i-1);
        cout << average ;

        return  0;
}
```

Declaration of Variables

- Example 2 : The following program is valid in C++ but would be illegal in C.(Example 2-1)

```
Void main( )
{
          int  x;              //L1
          x = 9;               //L2
          int  y;              //L3
          y = x+1;             //L1
}
```

Dynamic Initialization of Variables

- In C
  - A variable must be initialized using a constant expression.
  - And the C compiler would fix the initialization code at the time of compilation.

Dynamic Initialization of Variables

- In C++
  - It permits initialization of the variables at run time, which is referred to as *dynamic initialization*.
  - A variable can be initialized at run time using expressions at the place of declaration.
  - The following are valid initialization statements:

```
……
……
int  n = strlen(string);
……
float  area = 3.14159 * rad * rad;
```

Dynamic Initialization of Variables

- In C++
  - Both the declaration and the initialization of a variable can be done simultaneously at the place where the variable is used for the first time.
  - The following two statements

```
float  average;                 //declare where it is necessary
average = sum / i;
```

  - Can be combined into a single statement:

```
float  average = sum / i;//initialize dynamically at run time
```

Dynamic Initialization of Variables

- Dynamic initialization is extensively used in object-oriented programming.
- We can create exactly the type of object needed, using information that is known only at the run time.

Reference Variables

- C++ introduces a new kind of variable known as the ***reference*** variable.
- A reference variable provides an **alias** (alternative name) for a previously defined variable.
  - For example, if we make the variable sum a reference to the variable total, then sum and total can be used interchangeably to represent the variable.

Reference Variables

- A reference variable is created as follows:

```
data – type  &  reference – name = variable - name
```

  - Example

```
float  total = 100;
float  &  sum  = total;
```

  - Both the variables refer to the same data object in the memory.
  - Now, the statements `cout << total;` and `cout << sum;` both print the value 100.
  - The statement `total = total +10;` will change the value of both total and sum to 110.

Reference Variables

- Programme Reading (Example 2-5)

```
#include<iostream.h>
void main(){
        int  i=9;
        int&  ir=i;
        cout<<"i= "<<i<<"    "<<"ir="<<ir<<endl;
        ir=20;
        cout<<"i="<<i<<"    "<<"ir="<<ir<<endl;
        i=12;
        cout<<"i="<<i<<"    "<<"ir="<<ir<<endl;
        cout<<"Address of i is:"<<&i<<endl;
        cout<<"Address of ir is:"<<&ir<<endl;
}
```

Reference Variables

- A reference variable must be initialized at the time of declaration.
  - This establishes the correspondence between the reference and the data object which it names.
  - Examples

```
float  f;                  //L1
float  &fr;                //L2, error, fr is not initialized
float  &r1 = f;            //L3
float  &r2 = f;            //L4
float  &r3 = r1;           //L5
```

Reference Variables

- C++ assigns additional meaning to the symbol &.
  - Here, & is not an address operator.
  - The notation float& means reference to float.
  - Other examples are

```
int  n[10];
int & x = n[10];                //x is alias for n[10]
char  & a = ‘\n’;               //initialize reference to a literal
```

- The variable x is an alternative to the array element n[10].
- The variable a is initialized to the newline constant.
- This creates a reference to the otherwise unknown location where the newline constant \n is stored.

Reference Variables

- The following references are also allowed:
  - The following statements cause m to refer to x which is pointed to by the pointer p.

```
int  x;
int  *p = &x;
int  & m = *p;
```

  - The statement below creates an int object with value 50 and name n.

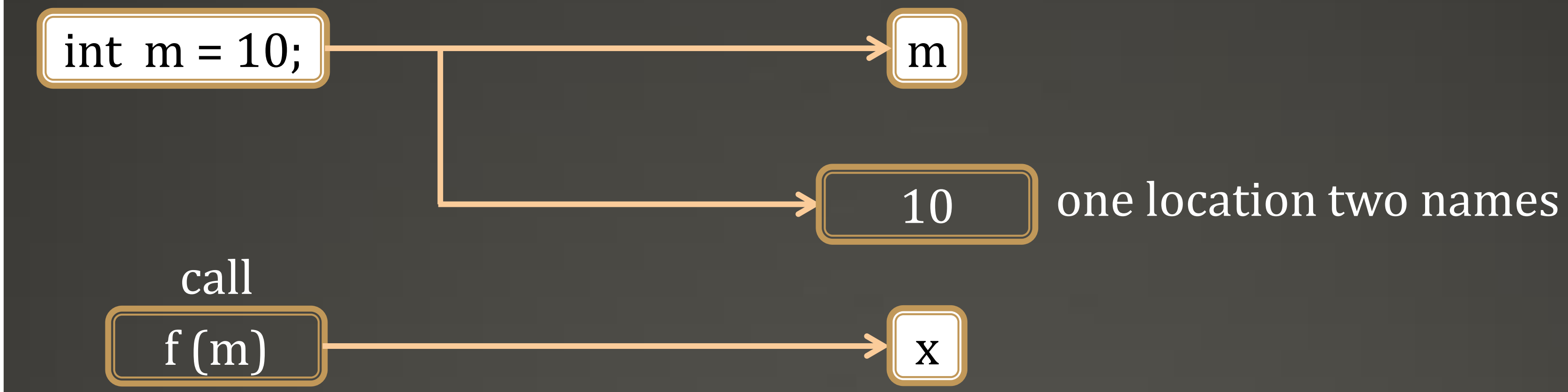
```
int  & n = 50;
```

Reference Variables

- A major application of reference variables in passing arguments to functions.

```
void  f(int & x)                    //uses reference
{
            x = x + 10;             //x is incremented; so also m
}
int  main( )
{
            int  m = 10;
            f (m);                  //function call
            ……
            ……
}
```

- When the function call f(m) is executed, the following initialization occurs: `int  & x = m;`
- Thus x becomes an alias of m after executing the statement `f(m);`
- Such function calls are known as ***call by reference***.



int m = 10;

m

10

one location two names

call

f (m)

x

Call by reference mechanism

- Since the variable x and m are aliases, when the function increments x, m is also incremented.
- The value of m becomes 20 after the function is executed.
- In traditional C, we accomplish this operation using pointers and dereferencing techniques.

Reference Variables

- The **call by reference** mechanism is useful in object-oriented programming because it permits the manipulation of objects by reference.
- And it eliminates the copying of object parameters back and forth.
- It is also important to note that references can be created not only by **built-in data types**, but also for **user-defined data types** such as structures and classes.

Operators in C++

- C++ has a rich set of operators.
- All C operators are valid in C++ also.
- In addition, C++ introduces some new operators.
    - For example
      the insertion operator << and the extraction operator >>

Operators in C++

- Other new operators are

::	Scope resolution operator
::*	Pointer-to-member declarator
->*	Pointer-to-member operator
.*	Pointer-to-member operator
delete	Memory release operator
endl	line feed operator
new	Memory allocation operator
setw	Field width operator

Operators in C++

- **Operator overloading**
  - C++ allows us to provide new definitions to some of the built-in operators.
  - That is, we can give several meanings to an operator, depending upon the types of arguments used.

Scope Resolution Operator

- Like C, C++ is also a block-structured language.
  - Blocks and scopes can be used in constructing programs.
  - The same variable name can be used to have different meanings in different blocks.
  - The scope of the variable extends from the point of its declaration till the end of the block containing the declaration.
  - A variable declared inside a block is said to be **local** to that block.

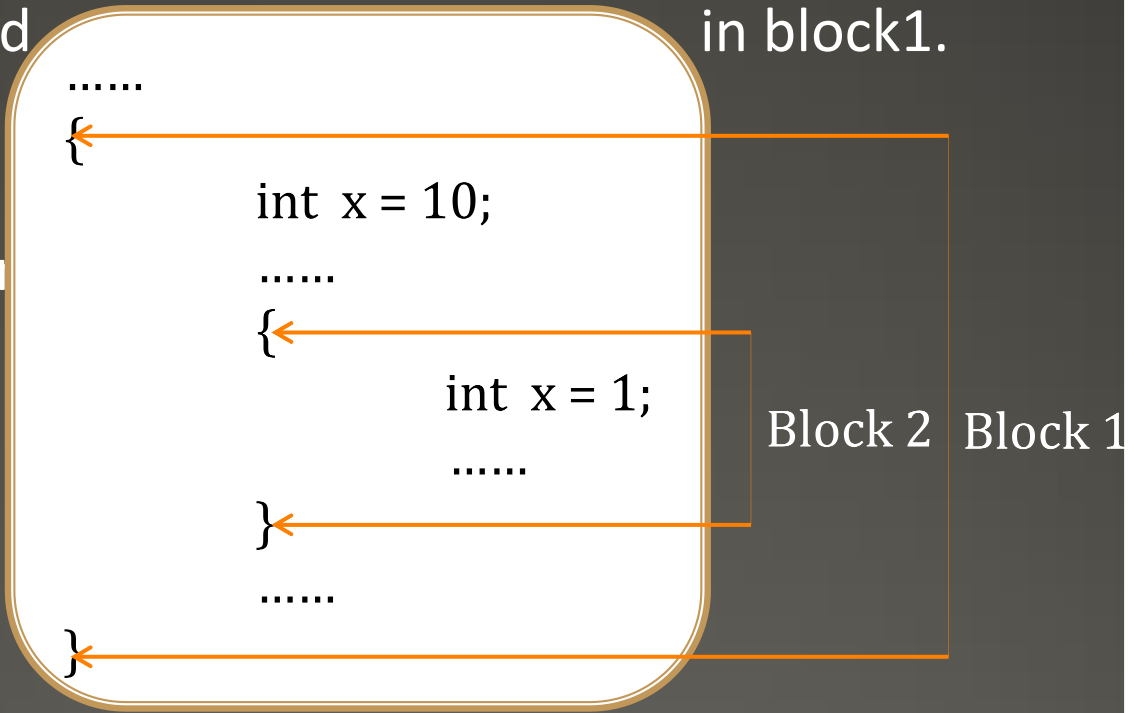
Scope Resolution Operator

- Consider the following segment of a program
  - The two declarations of x refer to two different memory locations containing different values.
  - Statements in the second block cannot refer to the variable x declared in the first block, and vice versa.

```
……
{
            int  x = 10;
            ……
}
……
{
            int  x = 1;
            ……
}
```

Scope Resolution Operator

- Blocks in C++ are often nested.
  - Block2 is contained in block1.
  - A declaration in an inner block **hides** a declaration of the same variable in an outer block.
  - Therefore, each declaration of x causes it to refer to a different data object.
  - Within the inner block, the variable x will refer to the data object declared therein.



```
......
{
        int  x = 10;
        ......
        {
                int  x = 1;
                ......
        }
        ......
}
```

Scope Resolution Operator

- In C, the global version of a variable cannot be accessed from within the inner block.
- Scope resolution operator
  - A new operator :: introduced by C++ to resolve this problem.
  - It can be used to uncover a hidden variable.
  - It takes the form `:: variable-name`
  - It allows access to the global version of a variable.
  - For example `:: count`

    It means the global version of the variable count (and not the local variable count declared in that block).

Scope Resolution Operator

- Example (Program 3.1)
  - The variable m is declared at three places, namely
    - Outside the main() function
    - Inside the main()
    - Inside the inner block
  - Note
    - ::m will always refer to the global m.
    - In the inner block, ::m refers to the value 10 and not 20.

Scope Resolution Operator

- A major application of the **scope resolution operator**
    - In the **classes** to identify the class to which a **member function** belongs.

Member Dereferencing Operator

- C++ permits us to define a **class** containing various types of data and functions as **members**.
- C++ also permits us to access the **class members** through pointers.
  - In order to achieve this, C++ provides a set of three **pointer-to-member** operators.

Member Dereferencing Operator

Member dereferencing operators

Operator	Function
::*	To declare a pointer to a member of a class
*	To access a member using object name and a pointer to that member
->*	To access a member using a pointer to the object and a pointer to that member

Memory Management Operator

- C uses **malloc()** and **calloc()** functions to allocate memory dynamically at run time.
- Similarly, it uses the function **free()** to free dynamically allocated memory.
- The **dynamic allocation** techniques is used when it is not known in advance how much of memory space is needed.

Memory Management Operator

- Although C++ supports these functions, it also defines two unary operators **new** and **delete** that perform the task of **allocating** and **freeing** the memory in a better and easier way.
  - Since these operators manipulate memory on the free store, they are also known as **free store operators**.

Memory Management Operator

- **New** & **Delete**
  - An object can be created by using **new**, and destroyed by using **delete**, as and when required.
  - A data object created inside a block with **new**, will remain in existence until it is explicitly destroyed by using **delete**.
  - Thus, the **lifetime** of an object is directly under our control and is unrelated to the block structure of the program.

Memory Management Operator

- **New**
  - The new operator can be used to create objects of any type.
  - It takes the following general form

```
pointer–variable = new  data-type;
```

  - **pointer-variable** is a pointer of type **data-type**.
  - The new operator allocates sufficient memory to hold a data object of type data-type and returns the address of the object.
  - The **data-type** may be any valid data type.
  - The **pointer-variable** holds the address of the memory space allocated.

Memory Management Operator

- New
  - Examples

```
p = new  int;
q = new  float;
```

  - where p is a pointer of type int and q is a pointer of type float.
  - Here, p and q must have already been declared as pointers of appropriate types.

Memory Management Operator

- **New**
  - Alternatively, we can combine the declaration of pointers and their assignments as follows

```
int  *p = new  int;
float  *q = new  float;
```

  - Subsequently, the statements

```
*p = 25;
*q = 7.5;
```

  - assign 25 to the newly created in object and 7.5 to the float object.

Memory Management Operator

- **New**
  - We can also initialize the memory using the new operator.

```
pointer–variable = new  data-type(value);
```

  - Here, value specifies the initial value.
  - Examples

```
int  *p = new  int(25);
float  *q = new  float(7.5);
```

Memory Management Operator

- **New**
  - **new** can be used to create a memory space for any data type including **user-defined** type such as arrays, structures and classes.
  - The general form for a one-dimensional array

```
pointer–variable = new  data-type[size];
```

  - Here, **size** specifies the number of elements in the array.
  - For example, the following statement creates a memory space for an array of 10 integers.

```
int  *p = new  int[10];
```

  - p[0] will refer to the first element, p[1] to the second element, and so on.

Memory Management Operator

- **New**
  - When creating multi-dimensional arrays with **new**, all the array size must be supplied.

```
array_ptr = new  int[3][5][4];      //legal
array_ptr = new  int[m][5][4];      //legal
array_ptr = new  int[3][5][ ];      //illegal
array_ptr = new  int[ ][5][4];      //illegal
```

  - The first dimension may be a variable whose value is supplied at runtime.
  - All others must be constants.

Memory Management Operator

- **Delete**
  - When a data object is no longer needed, it is destroyed to release the memory space for reuse.
  - The general form is

```
delete  pointer-variable;
```

  - The **pointer-variable** is the pointer that points to a data object created with **new**.
  - Examples

```
delete  p;
delete  q;
```

Memory Management Operator

- **Delete**
  - If we want to free a dynamically allocated array, we must use the following form of **delete**

```
delete  [size] pointer-variable;
```

  - The **size** specifies the number of elements in the array to be freed.
  - The problem with this form is that the programmer should remember the size of the array.
  - Recent versions of C++ do not require the size to be specified. For example `delete  [ ] p;`
    - It will delete the entire array pointed to by p.

Memory Management Operator

- **New**
  - What happens if sufficient memory is not available for allocation?
    - In such cases, like **malloc()**, **new** returns a null pointer.
  - Therefore, it may be a good idea to check for the pointer produced by **new** before using.

```
……
p = new  int;
if (!p)
{
            cout << “allocation failed \n”;
}
……
```

Memory Management Operator

- **New**
  - The **new** operator offers the following advantages over the function **malloc()**.
    1. It automatically computes the size of the data object. We need not use the operator **sizeof**.
    2. It automatically returns the correct pointer type, so that there is no need to use a type cast.
    3. It is possible to initialize the object while creating the memory space.
    4. Like any other operator, **new** and **delete** can be overloaded.

Manipulators

- Manipulators are operators that are used to format the data display.
- The most commonly used manipulators are **endl** and **setw**.

Manipulators

- The endl manipulator
    - When used in an output statement, it causes a linefeed to be inserted.
    - It has the same effect as using the newline character “\n”.

Manipulators

- The **endl** manipulator
  - For example, the statement would cause three line of output, one for each variable.
  - If we assume the values of the variables as 2597, 14, and 175 respectively, the output will
  - It is important to note that this form is not the ideal output.
  - It should rather appear as under (the numbers are **right-justified**).

```
......
cout << “m = ” << m << endl
        << “n = ” << n << endl
        << “p = ” << p << endl;
......
```

```
m = 2597
n = 14
p=  175
```

```
m = 2597
n =   14
p=   175
```

Manipulators

- The **setw** manipulator
  - The special form of output is possible only if we can specify a common field width for all the numbers and force them to be printed right-justified.
  - The **setw** manipulator does this job.
  - It is used as follows

```
cout << setw(5) << sum << endl;
```

  - The manipulator **setw(5)** specifies a field width 5 for printing the value of the variable sum.
  - This value is right-justified within the field shown below

		3	4	5

Manipulators

- Example (Program 3.2)
- Note
  - Character strings are also printed right-justified.

Manipulators

- We can also write our own manipulators as follows

```
#include  <iostream>
ostream & symbol(ostream & output)
{
            return output << “\tRs”;
}
```

  - The **symbol** is the new manipulator which represents **Rs**.
  - The identifier **symbol** can be used whenever we need to display the string **Rs**.

Type Cast Operator

- C++ permits explicit type conversion of variables or expressions using the **type cast operator**.
- Traditional C casts are augmented in C++ by a function-call notation as a syntactic alternative.

Type Cast Operator

- The following two statements are equivalent

```
(type-name) expression          //C notation
type-name (expression)          //C++ notation
```

- Examples

```
average = sum / (float) i;      //C notation
average = sum / float (i);      //C++ notation
```

- A type-name behaves as if it is a function for converting values to a designated type.

Type Cast Operator

- The function-call notation usually leads to simplest expressions.
- However, it can be used only if the type is an identifier.
  - For example `p = int  * (q);` is illegal.
  - In such cases, we must use C type notation. `p = (int  *) q;`
  - Alternatively, we can use **typedef** to create an identifier of the required type and use it in the function notation.

```
typedef  int * int_pt;
p = int_pt(q);
```

Type Cast Operator

- ANSI C++ adds the following new cast operators
  - const_cast
  - static_cast
  - dynamic_cast
  - reinterpret_cast

Expressions and Their Types

- **Expressions**
  - Combination of operators, constant and variables arranged as per the rules of the language.
  - It may also include function calls which return values.
  - An expression may consist of one or more operands, and zero or more operators to produce a value.

Expressions and Their Types

- Expressions may be of the following seven types
  - Constant expressions
  - Integral expressions
  - Float expressions
  - Pointer expressions
  - Relational expressions
  - Logical expressions
  - Bitwise expressions
- **Compound expressions**
  - An expression use combination of the above expressions.

Expressions and Their Types

- **Constant Expressions**
  - Consist of only constant values.
  - Examples

```
15
20 + 5 / 2.0
‘x’
```

Expressions and Their Types

- Integral Expressions
  - Produce integer results after implementing all the automatic and explicit type conversions.
  - Examples (where m and n are integer variables)

```
m
m * n -5
m * 'x'
5 + int(2.0)
```

Expressions and Their Types

- **Float Expressions**
  - After all conversions, produce floating-point results.
  - Examples (where x and y are floating-point variables)

```
x + y
x * y / 10
5 + float(10)
10.75
```

Expressions and Their Types

- Pointer Expressions
  - Produce address values.
  - Examples (where m is a variable and ptr is a pointer)

```
&m
ptr
ptr + 1
“xyz”
```

Expressions and Their Types

- ## Relational Expressions
  - Yield results of type **bool** which takes a value **true** or **false**.
  - Also known as **Boolean expressions**.
  - Examples

```
x <=y
a + b == c + d
m + n >100
```

  - When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared.

Expressions and Their Types

- **Logical Expressions**
  - Combine two or more relational expressions and produces **bool** type results.
  - Examples

```
a > b  &&   x==10
x==10  ||  y==5
```

Expressions and Their Types

- **Bitwise Expressions**
  - Used to manipulate data at bit level.
  - Basically used for testing or shifting bits.
  - Examples

```
x << 3              //Shift three bit position to left
y >> 1              //Shift one bit position to right
```

  - Shift operators are often used for multiplication and division by powers of two.

Special Assignment Expressions

- **Chained Assignment**

```
x = (y = 10);
     or
x = y =10;
```

  - First 10 is assigned to y and then to x.
  - A chained statement cannot be used to initialize variables at the time of declaration.
  - For instance, the following statement is illegal.

```
float  a = b = 12.34;                    //wrong
```

  - This may be written as

```
float  a = 12.34, b=12.34                //correct
```

Special Assignment Expressions

- Embedded Assignment

```
x = (y = 50) + 10;
```

  - (y = 50) is an assignment expression known as embedded assignment.
  - Here, the value 50 is assigned to y and then the result 50 + 10 = 60 is assigned to x.
  - The statement is identical to

```
y = 50;
x = y + 10;
```

Special Assignment Expressions

- ## Compound Assignment
  - Like C, C++ supports a **compound assignment operator** which is a combination of the assignment operator with a binary arithmetic operator.
  - For example, the simple assignment statement

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

  - may be written as

```
x += 10;
```

Special Assignment Expressions

- **Compound Assignment**
  - The operator += is known as **compound assignment operator** or **short-hand assignment operator**.
  - The general form of the compound assignment operator is

```
variable1  op= variable2;
```

  - where op is a binary arithmetic operator.
  - This means that

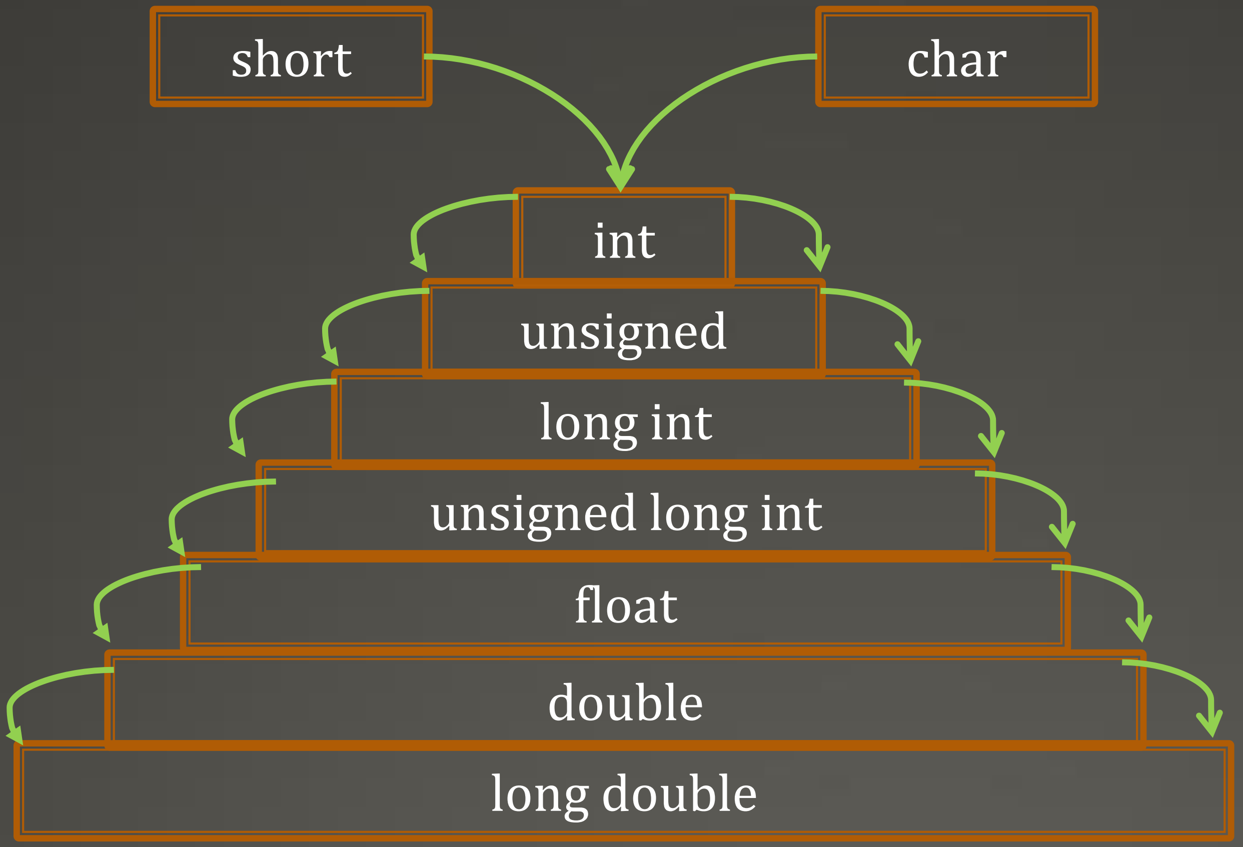
```
variable1 = variable1  op  variable2;
```

Implicit Conversions

- We can mix data types in expressions.
  - For example
  - `m = 5 + 2.75` is a valid statement.
- **Implicit** or **automatic conversion**
  - Wherever data types are mixed in an expression, C++ performs the conversions automatically.

Implicit Conversions

- **Implicit** or **automatic conversion**
  - When the compiler encounters an expression, it divides the expressions into sub-expressions consisting of one **operator** and one or two **operands**.
  - For a binary operator, if the operands type differ, the compiler converts one of them to match with the other, using the rule that the “smaller” type is converted to the “wider” type.
  - For example
    if one of the operand is an int and the other is a float, the int is converted into a float because a float is wider than an int.



short

char

int

unsigned

long int

unsigned long int

float

double

long double

Water-fall model of type conversion

Implicit Conversions

- **Integral widening conversion**
  - Whenever a **char** or **short int** appears in an expression, it is converted to an **int**.
- The implicit conversion is applied only after completing all integral widening conversions.

Operator Overloading

- C++ permits overloading of operators, thus allowing us to assign multiple meanings to operators.

Operator Overloading

- Actually, we have used the concept of overloading in C also.
  - For example
    - The operator * when applied to a pointer variable, gives the value pointed to by the pointer.
    - But it is also commonly used for multiplying two numbers.
    - The number and type of operands decide the nature of operation to follow.

Operator Overloading

- Examples of operator overloading
  - The input/output operators << and >>.
  - Although the built-in definition of the << operator is for shifting of bits, it is also used for displaying the values of various data types.
  - This has been made possible by the header file **iostream** where a number of overloading definitions for << are included.
  - Thus, the statement `cout << 75.86;` invokes the definition for displaying a double type value.
  - `cout << “well done”;` invokes the definition for displaying a char value.

Operator Precedence

- Although C++ enables us to add multiple meanings to the operators, yet their **association** and **precedence** remain the same.
  - For example, the multiplication operator will continue having higher precedence than the add operator.
- Table 3.5 (P64) gives the precedence and associativity of all the C++ operators.

Control Structures

- Sequence structure (straight line)
- Selection structure (branching)
- Loop structure (iteration or repetition)