Programming Paradigm

C++ Basic Concepts

Ashish Kr. Layek (CST, IIEST)

Simple Output Operations

```
/* Demonstration of output operation */
#include <iostream>
int main(void){
    std::cout << “Hello from Ashish K L” << std::endl;
    std::cout << “Hope you are ” << “doing well” <<std::endl;
    return 0;
}
```

Observe :

a) That the operator << can be used to place any simple type value (int, double, …) on to the variable of output stream

b) That the operator << has different semantics based on the context – this is also left shift operator on integer

c) That the result of the operation is a reference to an output stream variable – this allows cascading of the operator <<.

Simple Input / Output Operations

```
#include <iostream>
int main(void) {
    using std::cout;   using std::cin;    using std::endl;
    char name[20];  int age; char city[100];
    cout << "Greetings from Ashish K L" << endl;
    cout << "Enter your name : ";
    cin >> name;
    cout << "Enter your age and city : ";
    cin >> age >> city;
    cout << "Hello " << name << ", you are " << age << " year old from "
  << city << endl;
    return 0; }
```

<u>Observe :</u>

a) Operator >> requires l-value or reference to a variable on the right

b) There is no necessity of finding the address of the variable

Namespace

When a big program split across files, it is possible that the variables introduced in the global scope may clash resulting in linker time errors.

To avoid the clashes of the names in the global space, names are introduced in one or more named space by the programmer.

There is a namespace called **std** to which the names of all standard library functions and other declarations are introduced

For your further study

Constant

- Constant in C++ have a name, type as well as a value.
- Definition of constant can be placed in .h files.
- Named constants are used in the program for the following reasons
  - Improves readability and therefore decreases the effort for maintenance
  - Can be modified in code at only one place

- Constants could be used -
  - Constants in program body
  - Constant parameter in function
  - Constant members in object / structure
  - Constant object through which a method is invoked

Constant

Example

```
int x; int y; int z;
const int a = 10;    // a is a constant integer
// const int b;      // syntax error - no value provided
```

```
int * const p = &x; // p is a const pointer to integer
                    // should be initialized with definition
// p = &y;           // value of p can NOT be changed
*p = z;              // value of *p can be changed
```

```
const int * q = &x; // q is a pointer to an int which is a const
q = &y;              // value of q can be changed
// *q = z;           // value of *q can NOT be changed
```

Pointers and dynamic allocation

- C++ supports dynamic allocation (and deallocation) through two operators : new and delete
- The operator new requires a type as the operand. It allocates as much memory as required for the particular type and returns a pointer to the type specified
- The operator new also works on user defined types
- The operator new and delete can be overloaded (will be discussed later)

Example code :

```
int *p = NULL;
p = new int;
*p = 20;
delete p;
int *q = new int[5];
for(int i = 0; i < 5; i++){
        q[i] = i;
}
delete []q;
```

Remembering Alias

Alias example code

```
    int *p, *q;
    p = new int;
    *p = 20;
    q = p;    // q and p refer to the same location,
              // they are now aliases
    *q = 30; // even *p would have changed
```

Remembering Garbage

<u>Garbage example code</u>

```
    int *p = Null;
    p = new int;    // Space for an int is
                    // create/reserved
    *p = 20;
    p = new int;    // AHA ! Space for one more int is
                    // created, earlier value and
                    // location lost - it's a memory
                    // leak and also called garbage
                    // location without accesses
```

Remembering Dangling reference

Dangling reference example code

```
    int *p, *q;
    p = new int;
    *p = 22;
    q = p;    // q and p refer to the same location,
              // they are now aliases
    delete p; // space for int is released
    *q = 333; // AHA ! The pointer q points to a non-
              // existing location. This is called
              // dangling reference. This can causes
              // memory corruption
```

This typical issue happens in case of shallow copy

Reference Variable

<u>Example code :</u>

```
int a = 10; int x = 30;
int &b = a; // b is a reference to a; b is same as a.
            // no new integer location is allocated to b
            // whatever b is used, it automatically refers to a

cout << &a << " " << &b << endl;  // outputs same address
cout << b << endl; // outputs 10

int c = b; // c becomes 10
cout << c << endl; // outputs 10

c = 20;  // b remains unchanged
cout << b << endl; // still outputs 10
```

Reference Variable

Contd...

<u>Example code (continued from previous slide):</u>

```
b = c; // same as a = c
cout << a << endl; // outputs 20
// int &d; // Syntax error
// &b = x; // Syntax error

// int & arr[] = {a, x}; // Array of reference are not
     allowed
```

Few points on Pointers & Reference

- Pointers may be undefined or NULL; reference should always be associated with a variable

- Pointers may be made to point to different variable at different time; reference is always associated with the same variable throughout its life

- Pointers should be explicitly dereferenced; References are automatically dereferenced

Call by Reference or Reference Parameters

- 'C' provide only one mechanism of parameter passing – by Value.
  - Arguments are copied to the parameters and changes to the arguments are not reflected in the calling functions
  - If arguments should be changed by the called function, then the pointer (that is also value) to the arguments is passed. Thus simulating parameter passing by reference in C (but its is just a simulation).
  - In the called function, pointer has to be dereferenced to access the arguments. This is costly operation.

- 'C++' supports parameter passing by Value as well by Reference.
  - As reference parameters are just a copy of the actual parameter, there is no need of dereferencing, Reference parameters are automatically dereferenced.

Call by Reference

Example

```
void swap1(int x, int y) {
    int temp; temp = x; x = y; y = temp; }
void swap2(int * x, int * y) {
    int temp; temp = *x; *x = *y; *y = temp; }
void swap3(int & x, int & y) {
    int temp; temp = x; x = y; y = temp; }

int main() {
    int a = 10, b = 20;
    swap1(a, b);     // call by value - changes not reflect
    swap2(&a, &b);   // call by value with pointer -
                     // changes reflects
    swap3(a, b);     // call by reference - changes reflects
    return 0; }
```

Call by Reference

Few points

- Less overhead in terms of time & space at runtime compared to parameter passing by value
- Can avoid problems resulting in shallow copy of parameter passing by value – this happens when a structure or an object has a pointer as a member
- Can be passed const reference to avoid changing the argument in the function – this is in case arguments are not required to change in the called function
- It is also possible to return by reference when an lvalue is required in the calling function
- It is efficient w.r.t. passing by pointer for a complex structure as dereferencing is not required

Conclusion : Always preferable to pass by reference

Return by Reference

If you return a reference from a function, you must take the same case as if you are return a pointer from a function

- Whatever the reference is connected to shouldn’t go away when the function returns
- Otherwise you will be referring to unknown memory location

```
int * f (int *x) {
    (*x)++;
    return x; // Safe
}
```

```
int & g (int & x) {
    x++;
    return x; // Safe
}
```

```
int main() {
    int a = 20;
    f(&a); // Ugly (but
explicit)
    g(a); // Clean (but
hidden)
}
```

Function Overloading - Static Polymorphism

It is possible to have more than one function with the same name where the calls are resolved at compile time based on matching the arguments to the parameters

Points on function overloading :

- ➢ More than one function with the same name
- ➢ Function call resolution is a compile time phenomenon and there is no extra overhead at runtime
- ➢ Functions should be declared in the same scope
- ➢ Resolution based on the number, type and order of arguments in the function
- ➢ Resolution does not depends on the result type

Function Overloading

An example

```
void display(char * name) {
    cout << "Name " << name << endl;
}
void display(char * name, int age){
    cout << "Name " << name << " Age " << age << endl;
}
int main() {
    char name[10] = "Aman";
    int age = 10;
    display(name, age); // Outputs : Name Aman Age 10
    display(name); // Name Aman
    return 0;
}
```

Default Parameter

There are many functions which have a large number of parameters. The user may not be interested in providing all these arguments in the call and he might be happy if the system can choose some default appropriate values for some of these parameters.

Example code

```
void display(const char * = "India", int = 72);
int main() {
    display("Bangladesh", 48); // Calling with two arguments
    display("India"); // Calling with one argument
    display(); // Calling with no argument
    return 0;
}
void display(const char * name, int age) {
    cout << "Country " << name << " Age " << age << endl;
}
```

Default Parameter

Few points

- The parameters should have a reasonable default
- It is a way of expanding the function
- Default value(s) are specified as part of the declaration
- Can be specified for functions whose source codes are not available – function writer may not know anything about these defaults
- Only rightmost parameters can be default – all parameters can also be default
- Leftmost arguments should be specified

Inline function

- Inline functions are used to reduce the overhead of normal function call
- It's a request to the C++ compiler that inline substitution of the function body is to be preferred to the usual function call implementation
- Suggestion could be ignored based on the compilers discretion
- Unlike Macro, it does not leave any side effect due to expansion of code
- The compiler checks for the proper use of the function argument list and return value – macro is incapable of.

Example Code :

```
#define MC_SQUARE(X) X*X
inline int fn_square(int x) { return x*x;  }
int a = b = 3;
int i = MC_SQUARE(++a);     // results in 20
int j = fn_square(++b);     // results in 16
```

Inline function Cont…

- Any function defined within a class body is automatically inline
- For a non-class inline function, the function name is preceded with inline keyword
  - It must include function body with the declaration, otherwise compiler will treat it as a ordinary function.
  - Put inline definition in a header file
- Compiler keeps the inline function type and body in its symbol table. When inline function called correctly, compiler substitute the function body from the function call – thus eliminating the overhead
- The inline function does occupy space, but if the function is small, this can actually take less space than the code generated from a ordinary function call (pushing arguments on the stack and doing the function call)

Template function

There are many instances where the structure of the code (logic) does not change with the input type. In such case it is possible to write generic routines where the type information can be passed as parameter in the calling code.

Example code :

```
template <typename T>  // T is a type parameter
void add(T &x, T &y) {
    T temp = x + y;
    cout << temp << endl;
}
int main() {
    int a = 10, b = 20;
    double p = 2.4, q = 4.5;
    add(a, b); // Causes compilation type instantiation of the
               // template function with type T = int.
    add<double>(p, q); // Another instance of the template
                       // function with type T = double.
}
```

Template function

Few points

- ❑ Mechanism to generate functions at compile type
- ❑ No extra overhead at run time
- ❑ Used to making functions generic
- ❑ Used when behavior/algorithms of the functions are same
- ❑ Gets implicitly instantiated by the call
- ❑ Can also be explicitly instantiated
- ❑ Instantiated for each type only once