Structure of C++ Program

The code is organized as follows :

- The class declarations are placed in a header file.
- The definitions of member functions go in another file.
- The main program is placed in the third file which *includes* the other two files as it will be using the classes created in those files.

Generating the Executable

- The following command compiles and links a program to generate an executable :

  c++ try.cpp

  This command generates a file called 'a.out' which is an executable file.

- The following only compiles the program file to generate an object file :

  c++ -c try.cpp

  This generates a file 'try.o' which is the object file corresponding to the program file 'try.cpp'.

  This object file can then be linked by the *c++* command to get the executable file 'a.out'.

- The default executable produced by *c++* is 'a.out' as the executable. But the executable name can be changed by using the *o* flag with *c++*

  c++ try.o -o try.exe

  c++ try.cpp -o try.exe

  This command produces try.exe as the executable file.

Data Types and Control Structures

Data Types :
The data types are the same as in C.

There is a new kind of variable that is introduced in C++. It is the ***reference*** variable.

Example :

```
int     i = 10 ;
int     & x = i ;
x = 20 ;
cout<<"value of i is "<<i<<endl ;
```

Output : *value of i is 20*

A reference variable :

- ❖is another name for an already created variable
- ❖must be initialized at the time of declaration
- ❖can be created for built – in as well as user defined data types
- ❖**References are compiled as pointers which are implicitly dereferenced at each use** – involve hidden dereference operations that are costly

Application of reference variables in argument passing :

```
void f(int &i)
{ i = 10 ;}

main( )
{
        int     k = 12 ;
        f(k) ;
        cout << k ;
}
```

Reference variables to arrays :

(An array is a **constant pointer**, it cannot be reset to hold another address)

```
void f(int *const&  x){}
void g(char *const&x){}

main() {
   int i[10] ;
   int * const p = i ;
   int * const &rp = i ;
   f(i) ;


   char arr[20] ;
   g(arr) ;
}
```

Using the *const* qualifier :

- If a function argument is declared as const then that argument can not be modified within the function. Else a compiler error is generated.

  Example :

```
void  func( const int &x, int y) {
x = 10 ; // generates an error
y += x ; }
```

- constant pointer (must be initialized during declaration) :

```
char  * const ptr1 = "HELLO" ;
// the address in ptr1 can not be modified henceforth
```

i.e., following statement is illegal :

```
ptr1 = new char[10] ;
```

- pointer to a constant :

```
int  const  * ptr2 = &m ;//here int and const can be interchanged without causing any effect
// ptr2 can point to any variable of matching type, but the contents of what it
points to can not be changed.
```

i.e., following is illegal :

```
*ptr2 = 80 ;
```

And following is legal :

```
ptr2 = &k ;
```

The following code depicts the difference between a 'constant pointer' and 'pointer to a constant' :

```
main() {
   int x = 10, y = 9 ;
   int *   const p1 = &x ;
   int const *  p2 = &x ;

   *p1 = 77 ;
   *p2 = 55 ;

   p1 = &y ;
   p2 = &y ;
}
```

Using *const* for overloading :

```
#include <iostream.h>

void f(char *ch) {cout<<"no const\n" ;}
void f(const char *ch) {cout<<"with const\n" ;}

main()
{
   char ch = 'c' ;
   char *cptr = &ch ;
   const char *cc = "hell" ;
   f(cptr) ;
   f(cc) ;
}
```

Thanks and Regards
Tusar Mondal
+91- 8585069236