Moving from C to C++

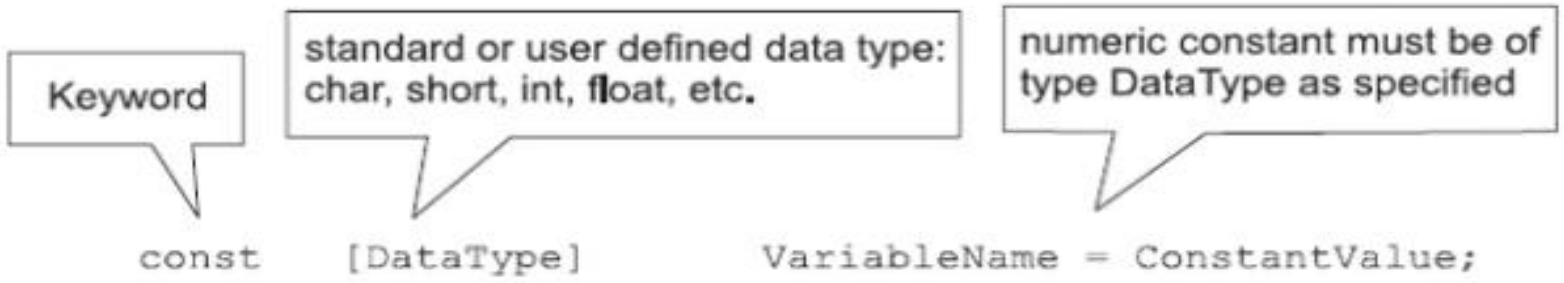
10th August 2020

Literals, Constant, Qualifier

- Literals are constants to which symbolic names are associated for which the purpose of readability and ease of handling standard constant values
- C++ provides three ways to define constants
  - #define preprocessor directives
  - Enumerated datatypes
  - const keyword

Constant Variable

- Constant value does not change throughout the life of the program



const float PI=3.1352

const int TRUE=1;

const char *book_name="oops with C++";

Note: if try to modify constant-type variable leads to compilation error

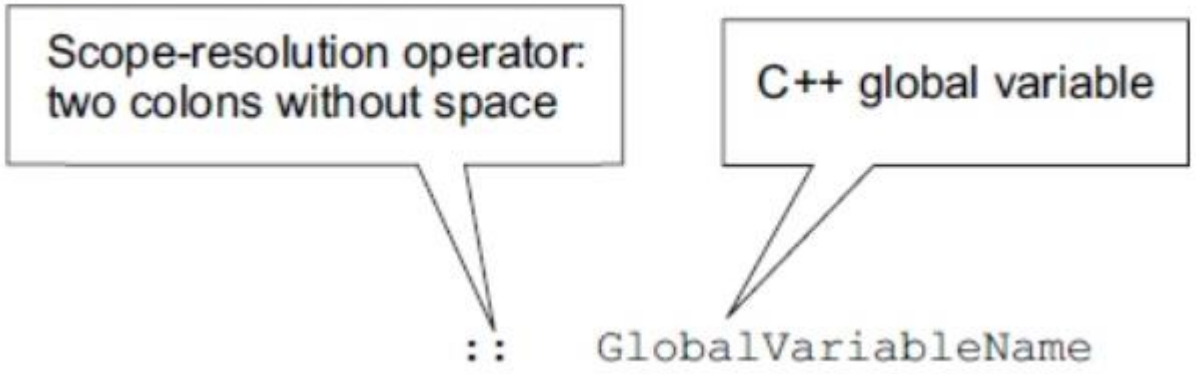
Constant Variable

```
#include<stdio.h>
#include<string.h>
void display( const char *msg)
{
   printf("%s",msg);
   strcpy(msg,"world");
}
int main()
{
   char string[10];
   strcpy(string,"hello");
   display(string);
   printf("%s",string);
}
//with warning gives the output in C
```

```
#include<iostream>
#include<string.h>
using namespace std;
void display( const char *msg)
{
   cout<<msg;
   strcpy(msg,"world"); //produces compilation error
}
int main()
{
   char string[10];
   strcpy(string,"hello");
   display(string);
   cout<<endl<<string;
}
```

Scope resolution operator(::)

- Used for accessing global variable if local and global variables are of same name
- Syntax is as follows



- The scope resolution

operator permits a program

to reference an identifier in the global scope that has been hidden by another identifier with the same name in local scope

Example

```
#include<iostream>
using namespace std;
int m=20;
int main()
{
   int m=30;
   cout<<"local="<<m<<endl;
   cout<<"global="<<::m<<endl;
   cout<<"local+global="<<m+::m<<endl;
   return 0;

}
```

output

```
local=30
global=20
local+global=50
```

Variables aliases Reference variable

- Reference variable is the other name of the variable
- It behaves like both value variable and pointer variable
- In program code it is used similar as value variable and has an action as pointer variable
- The reference variable enjoys the simplicity of value variable and the power of pointer variable
- It does not support the flexibility of the pointer variable
- Unlike pointer variable when reference variable bound to a variable then its binding can not changed
- Syntax

  Datatype &Referencevariable = Variable

Example

```
1. char & chl = ch;          // chl is an alias of char ch
2. int & a = b;              // a is an alias of int b
3. float & x = y;
4. double & height = length;
5. int &x = y[100];          // x is an alias of y[100] element
6. int n;
   int *p = &n;
   int &m = *p;
7. int &num = 100;           // invalid
```

This statement causes compilation error; constants cannot be made to be pointed to by a reference variable. Hence, the rule: *no alias for constant value.*

Example Reference variable

```
#include<iostream>
using namespace std;
int main()
{
   int a=1, b=2, c=3;
   int &z=a; //z alias for a
   cout<<"a="<<a<<" b="<<b<<" c="<<c<<" z="<<z<<endl;
   z=b; //changes value a to b
   cout<<"a="<<a<<" b="<<b<<" c="<<c<<" z="<<z<<endl;
   z=c; //changes to c
   cout<<"a="<<a<<" b="<<b<<" c="<<c<<" z="<<z<<endl;
   cout<<"&a="<<&a<<" &b="<<&b<<" &c="<<&c<<" &z="<<&z<<endl;
}
```

output

a=1 b=2 c=3 z=1

a=2 b=2 c=3 z=2

a=3 b=2 c=3 z=3

&a=0x61fe14
&b=0x61fe10
&c=0x61fe0c
&z=0x61fe14

Example

```
#include<iostream>
using namespace std;
int main()
{
   int n=100;
   int *p=&n;
   int &m=*p;
   cout<<"n = "<< n<<" m = "<< m<<" *p = "<< *p<<endl;
   int k=5;
   p=&k;
   k=200;
   cout<<"n = "<< n<<" m = "<< m<<" *p = "<< *p<<endl;
}
```

output

```
n = 100 m = 100
*p = 100
n = 100 m = 100
*p = 200
```

Swap using pointer and reference

```
#include<iostream>
using namespace std;
void swap(int *x, int *y)
{
   int temp;
   temp=*x;
   *x=*y;
   *y=temp;
}
int main()
{
   int a,b;
   cout<<"Enter value of a & b"<<endl;
   cin>>a>>b;
   cout<<"Before swap a="<<a<<" b="<<b<<endl;
   swap(&a,&b);
   cout<<"After swap a="<<a<<" b="<<b<<endl;
   return 0;
}
```

```
#include<iostream>
using namespace std;
void swap(int &x, int &y)
{
   int temp;
   temp=x;
   x=y;
   y=temp;
}

int main()
{
   int a,b;
   cout<<"Enter value of a & b"<<endl;
   cin>>a>>b;
   cout<<"Before swap a="<<a<<" b="<<b<<endl;
   swap(a,b);
   cout<<"After swap a="<<a<<" b="<<b<<endl;
   return 0;

}
```

Enumerated Types or Enum

- **Enumerated type** (enumeration) is a user-defined data type which can be assigned some limited values. These values are defined by the programmer at the time of declaring the enumerated type.
- It can be created in two types:-
  - It can be declared during declaring enumerated types, just add the name of the variable before the semicolon.
  - Beside this, we can create enumerated type variables as same as the normal variables.
- Syntax

  enum enumerated-type-name{value1, value2, value3…..valueN};

  enum color{red, orange, yellow, green};

  enumerated-type-name variable-name = value;

  enum compass{ north = 0, east = 90, south = 180, west = 270};

Example

```
//Prg15.cpp
#include<iostream>
using namespace std;
enum compass{ north = 0, east = 90, south = 180, west = 270};
enum color{red, orange, yellow, green};
enum answer{ yes, no};
int main(void)
{
        int answer;
        color light;
        compass direction;
        answer = yes;
        if(answer == yes)
        {
                direction = south;
                light = yellow;
        }
        cout<<direction<<endl;
        cout<<light;
}
```

Function

- A function provides a convenient way of packaging a computational steps, so that it can be used as often as required.
- A function definition consists of two parts:
  - interface and body.
- The interface of a function (also called its prototype) specifies how it may be used.
- It consists of three entities:
  - The function name. This is simply a unique identifier.
  - The function parameters (also called its signature). This is a set of zero or more typed identifiers used for passing values to and from the function.
  - The function return type. This specifies the type of value the function returns. A function which returns nothing should have the return type void.

Inline Functions

- In C++ you can have true inline functions. C++ compiler may refuse to expand the function if it is considered too large
- Inline functions make the executable code larger but execution time is faster as there is no house keeping work for function call
- Overuse of inline functions in large program is highly discouraged
- Syntax

```
inline ReturnType FunctionName(parameters)
{
//Function body
}
```

Inline Function Example

```
inline int min(int x, int y)
{
if(x < y)
    return x;
else
return y;
}
inline int max(int x, int y)
{
    if(x > y)
            return x;
    else
            return y;
}
```

```
int main(void)
{
     int x = 7, y = 10;
    cout << “Minimum is “ << min(x,y) << endl;
    cout << “Maximum is” << max(x, y) << endl;
     return 0;
}
```

In case of function call it is to be expanded inline rather than called, precede its definition with the inline keyword.

For example, in this program, the function max() and min() are expanded in line instead of called.

each time a function is called, a significant amount of overhead is generated by the calling and return mechanism, which is avoided incase of inline function

Inline function

- Some of the situation where inline expansion may not work
- For function returning values, if switch, goto or a loop exists
- For functions not returning values , if a return statement exists
- If a function contain static variables
- If inline functions are recursive

Default Arguments

- Default argument is a programming convenience which removes the burden of having to specify argument values for all of a function's parameters.
- C++ allows us to call a function without specifying all its arguments
- In such cases, the function assigns a default value to the parameter which does not have a matching argument in the function call
- Default values are specified when the function declared.
- The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values

Example : default arguments

```
#include <iostream>
using namespace std;
void total(int x1, int x2=10, int x3 = 100); // prototype
void total(int x1, int x2, int x3)
{
   cout << “Total = “ << x1 + x2 + x3 << endl;
}
int main(void)                          Result of run
{
   total(5);                            Total = 115
   total(5, 50);                        Total =155
   total(5, 50, 500);                   Total = 555
   return 0;
}
```

default arguments

- A default argument is checked for type at the time of declaration and evaluated at the time of call
- Only the trailing arguments can have default values
- The default values are added from right to left
- In middle default vales can’t be assigned

```
int total( int i, int j=5, int k=10) ;     //legal
int total( int i=5, int j) ;               //illegal
int total( int i=0, int j, int k=10) ;  //illegal
int total( int i=2, int j=5, int k=10) ;  //legal
```

Function overloading

- Function overloading is the process of using the same name for two or more functions.
- The secret to overloading is that each redefinition of the function must use
  - either different types of parameters
  - or a different number of parameters.
- It is only through these differences that the compiler knows which function to call in any given situation.
- For example, the following program overloads square() by using different types of parameters.

Function overloading Example

```
void square(int n)
{
   cout << “square of integer ” << n << “ = “ << n*n << endl;
return n*n;
}
void square(double n)
{
   cout << “square of double “ << n << “ = “ << n*n << endl;
}
void square(char ch)
{
   cout <<“square of character “ << ch <<“ = “ <<ch <<ch << endl;
}
```

```
#include <iostream>
using namespace std;
int main(void)
{
   int n=5;

   double d=3.5;
   square(n);

   square(d);

   square(‘A’);

   return 0;
}
```

Ambiguities in name overloading

```
int max(int a, int b)
{
   if(a>b)
            return a;
else
            return b;
}
double max(double a, double b)
{
   if(a>b)
            return a;
else
            return b;
}
```

```
int main(void)
{
    double m = max(5, 12.75);// error
     int x=max(2,3);
     cout<<x<<endl;
    double m = max(5.0, 12.75);
   double m = max((double)5, 12.75);
    cout << m << endl;
    return 0;
}
```

No function max is defined for mixed arguments, thus, either the int may be type casted to double, or the double may be type casted to int.

there is an ambiguity and the compiler will generate error in this case.

It should be recoded; in this case the function with int arguments may be dropped altogether.

Ambiguities in function overloading

- As mentioned, the key point about function overloading is that the functions must differ in regard to the types and/or number of parameters.
- Two functions differing only in their return types cannot be overloaded. For example, this is an invalid attempt to overload myfunc()

```
int myfunc(int i);
float myfunc(int i);
```

- // Error: differing return types are insufficient when overloading
- Sometimes, two function declarations will appear to differ, when in fact they do not. For example, consider the following declarations.

```
void f(int *p);
void f(int p[]);
```

// error, *p is same as p[]

Heap Management

- C++ places great store in the heap. There are four operators for heap management
  - new allocates an object on the heap
  - new[n] allocates arrays of n objects on heap
  - delete deallocates objects on heap
  - delete[ ] deallocates arrays of objects on heap

new & delete

```
int main(void)
{
   double *p;
   int *q;
   p = new double;
   q =new int [15];
   // rest of the program
   delete p;
   delete q[ ];
  //delete [ ] q;
   return 0;
}
```

New [ ] & delete [ ]

```
const int MAX = 10;
int main(void)
{
    int *array;
    array = new int[MAX];
    for(int i=0; i<MAX; i++)
    {
            array[i] = i * i;
            cout << “Square of  “ << i << “ is “ << array[i] << endl;
    }
Delete [ ] array;
return 0;
}
```

Control Structure in C++

- **Control Structures** are just a way to specify flow of control in programs.
- There are three basic types of logic, or flow of control, known as:
  - Sequence logic, or sequential flow
  - Selection logic, or conditional flow
  - Iteration logic, or repetitive flow
- Selection logic
  - If-else
  - switch
- Iteration
  - for loop
  - do while loop
  - while loop

Structure in C++

- **Structures in C++** are user defined data types which are used to store group of items of non-similar data types.
- C++ allows function as part of the structure
- Structures in C++ can contain two types of members:
- **Data Member**: These members are normal C++ variables. We can create a structure with variables of different data types in C++.
- **Member Functions**: These members are normal C++ functions. Along with variables, we can also include functions inside a structure declaration.
- By default all the members of the structure are public members
- We can define private, public and protected member variables and function inside structure.

Structure

Syntax

```
struct StructureName
{
  public:
    // data and functions
  private:
    // data and functions
  protected:
    // data and functions
};
```

Default all member are public

```
//prg14.cpp
#include<iostream>
using namespace std;
struct  Student
{
        int roll;
        int age;
        int marks;
        void getDetails()
        {
                cout<<"Enter Roll";
                cin>>roll;
                cout<<"Enter Age";
                cin>>age;
                cout<<"Enter Mark";
                 cin>>marks;
        }
void printDetails()
        {
           cout<<"Roll = "<<roll<<"\n";
           cout<<"Age = "<<age<<"\n";
           cout<<"Marks = "<<marks<<"\n";
        }
};
int main()
{
  Student s1;
  s1.getDetails();
  s1.printDetails();
  s1.roll=10;
  s1.age=20;
  s1.marks=30;
  s1.printDetails();
  return 0;
}
```

```
#include<iostream>
using namespace std;
struct  Student
{
        private:
        int roll;
        int age;
        int marks;
        public:
        void getDetails()
        {
                roll=10;
                //cout<<"Enter Roll";
                //cin>>roll;
                cout<<"Enter Age";
                cin>>age;
                cout<<"Enter Mark";
                 cin>>marks;
        }
void printDetails()
                {
                   cout<<"Roll = "<<roll<<"\n";
                   cout<<"Age = "<<age<<"\n";
                   cout<<"Marks = "<<marks<<"\n";
                }
};
int main()
{
  Student s1;
  s1.getDetails();
  s1.printDetails();
  s1.roll=10; //error
  s1.age=20; //error
  s1.marks=30;//error
  s1.printDetails();
  return 0;
}
```

```
#include<iostream>
using namespace std;
struct  Student
{
        private:
        int roll;
        int age;
        int marks;
        void getDetails();
        void printDetails();

};
 void Student:: getDetails()
{
    cout<<"Enter Roll";
     cin>>roll;
     cout<<"Enter Age";
     cin>>age;
     cout<<"Enter Mark";
     cin>>marks;
}
```

```
void Student::printDetails()
{
        cout<<"Roll = "<<roll<<"\n";
        cout<<"Age = "<<age<<"\n";
        cout<<"Marks = "<<marks<<"\n";
}
int main()
{
  Student s1;
  s1.getDetails(); //error
  s1.printDetails();//error
  s1.roll=23;//error
return 0;
}
```