Functions in C++

function

- Function is a group of valid statements combined together to perform a well defined task
- Components to add for a user defined-function
  - The function declaration
  - The calls to the function
  - The function definition

Function example

```
//demonstrates function call
#include <iostream>
using namespace std;

void repchar(char, int); //function declaration

int main() {
    repchar('-', 43); //call to function
    cout << "Data typ          Range" << endl;
    repchar('=', 23);
    cout << "char          -128 to 127" << endl;
    cout << "short         -32,768 to 32,767" << endl;
    cout << "int           System dependent" << endl;
    cout << "double        -2,147,483,648 to 2,147,483,647" << endl;
    repchar('-', 43);
    char chin; int nin;
    cout << "Enter a character: ";
    cin >> chin;
    cout << "Enter number of times to repeat it: ";
    cin >> nin;
    repchar(chin, nin);
    return 0;
}
```

```
void repchar(char ch, int n) {
    for(int j=0; j<n; j++)
       cout << ch;
    cout << endl;
}
```

Pass by value

- When constants or variables are passed to it, the function creates new variables to hold the values of these variable arguments.
- The function gives these new variables the names and data types of the parameters specified in the declarator:
  - In example : ch of type char and n of type int.
- Passing arguments in this way, where the function creates copies of the arguments passed to it, is called passing by value.

Passing a structure variable

```
// demonstrates passing and returning a
structure
#include <iostream>
using namespace std;

struct Distance {
       int feet;
       float inches;
};

Distance addDist(Distance, Distance);
void dispDist(Distance);

int main() {
Distance d1, d2, d3;
    cout << “\nEnter feet: “; cin >> d1.feet;
    cout << “Enter inches: “; cin >> d1.inches;

    cout << “\nEnter feet: “; cin >> d2.feet;
    cout << “Enter inches: “; cin >> d2.inches;
    d3 = addDist(d1, d2);
    cout << endl;
    dispDist(d1); cout << “ + “;
    dispDist(d2); cout << “ = “;
    dispDist(d3); cout << endl;
    return 0;
}
```

```
// adds two structures of type Distance, returns
sum
Distance addDist( Distance dd1, Distance
dd2 ) {
    Distance dd3;
    dd3.inches = dd1.inches + dd2.inches;
    dd3.feet = 0;
    if(dd3.inches >= 12.0) {
       dd3.inches -= 12.0; //by 12.0 and
       dd3.feet++; //increase feet
    }
    dd3.feet += dd1.feet + dd2.feet; //add the
feet
    return dd3; //return structure
}


void dispDist( Distance dd ) {
    cout << dd.feet << “\’-” << dd.inches << “\””;
}
```

Reference Arguments

- Passing arguments by value is useful when the function does not need to modify the original variable in the calling program.
- In passing arguments by reference, a reference to the original variable in the calling program is passed.
- An important advantage of passing by reference is that the function can access the actual variables in the calling program.
- Among other benefits, this provides a mechanism for passing more than one value from the function back to the calling program.

Example

```
// demonstrates passing by reference
#include <iostream>
using namespace std;
int main() {
    void intfrac(float, float&, float&); //declaration
    float number, intpart, fracpart; //float variables
    do {
        cout << “\nEnter a real number: “;
        cin >> number;
        intfrac(number, intpart, fracpart);
        cout << “Integer part is “ << intpart
        << “, fraction part is “ << fracpart << endl;
    } while( number != 0.0 );
    return 0;
}

// finds integer and fractional parts of real number
void intfrac(float n, float& intp, float& fracp) {
    long temp = static_cast<long>(n);
    intp = static_cast<float>(temp);
    fracp = n - intp;
}
```

Default arguments

- Parameters can be assigned default values.
- Parameters assume their default values when no actual parameters are specified for them in a function call.
- A default argument is type checked at the time of function declaration and evaluated at the time of call
- Default arguments may be provided for trailing arguments only

Example: default arguments

```
// Find the sum of numbers in a range of values between “lower” and “upper”
using increment “inc”
int sum(int lower,int upper=100,int inc=1){
          int sum=0;
          for(int k=lower; k<=upper; k+= inc)
                sum += k;
return sum;
}
Int main(){
         cout<<sum(1);
         cout<<sum(1, 10);
         cout<<sum(1, 10, 2);
         return 0;
}
```

we are going left to right

recursion

- Recursion involves a function calling itself.
- Recursion is much easier to understand with an example than with lengthy explanations
- Each version of the recursive function stores its own value of variables while it’s busy calling another version of itself.
- When a recursive function completes its execution, it returns to the previously called version of itself
- Every recursive function must be provided with a way to end the recursion. Otherwise it will call itself forever and crash the program.
- It is not true that many versions of a recursive function are stored in memory while it’s calling itself. Each version’s variables are stored, but there’s only one copy of the function’s code.

Inline function

- While a function may help to save memory space, its call and return process requires some extra time.
- There must be an instruction
  - for the jump to the function instructions
  - for pushing arguments onto the stack and removing them,
  - for restoring registers
  - to return to the calling program and dealing with return values
- All these instructions slow down the program.

Inline function

- Making a short section of code into a function may impose penalty just as much as a larger function.
- One solution is to simply repeat the necessary code in your program
- However, the repeated code lose the benefits of program organization and clarity.
- The solution to this issue is the inline function.

Inline function

- Inline functions is a normal function in the source file but compiles into inline code instead of into a function.
- The function is shown as a separate entity.
- However, when the program is compiled, the inline function is actually inserted into the program wherever a function call occurs.
- Functions that are very short, say one or two statements, are candidates to be inlined

Elements violating inline property

- A recursive call to the function
- A loop inside function
- Functions whose address is referenced somewhere
- Virtual functions (there are some exceptions)
- A large number of statements
- The last case does not generate any error. Nevertheless, the inline property is violated

Example

```
// demonstrates inline functions
#include <iostream>
using namespace std;

inline float lbstokg(float pounds) {
    return 0.453592 * pounds;
}

int main() {
    float lbs;
    cout << “\nEnter your weight in pounds: “;
    cin >> lbs;
    cout << “Your weight in kilograms is “ << lbstokg(lbs) << endl;
    return 0;
}
```

Function Calls on the Left of the Equal Sign

- A function that returns a reference, on the other hand, is treated as if it were a variable.
- It returns an alias to a variable, namely the variable in the function’s return statement.
- Thus it can be used on the left side of an equal sign

```
int main() {
        int i=2,j=3;
        max(i,j)= -30;
        cout<<“i=“<< i << ‘\t’ <<“j=“ << j << ‘\t’ <<endl;
}
int& max(int &x, int &y){
             return x>y ? x:y;
}
```

i = 2, j = -30

Function overloading

- We can use same function name to create funcions that perform a variety of different task
- Also called function polymorphism
- An overloaded function appears to perform different activities depending on the kind of data sent to it.
- It performs one operation on one kind of data but another operation on a different kind.
- Two variations
  - Function overloaded with different number of arguments
  - Function overloaded with different kind of arguments

Function Overloading with Different Numbers of Arguments

```
// demonstrates function
overloading
#include <iostream>
using namespace std;

void repchar();
void repchar(char);
void repchar(char, int);

int main() {
    repchar();
    repchar(‘=’);
    repchar(‘+’, 30);
    return 0;
}

// repchar() : displays 45
asterisks
void repchar() {
    for(int j=0; j<45; j++)
        cout << ‘*’;
    cout << endl;
}
```

```
// repchar() : displays 45 copies of specified
character
void repchar(char ch) {
    for(int j=0; j<45; j++) // always loops 45
times
        cout << ch;
    cout << endl;
}
// repchar() : displays specified number of
copies
// of specified character
void repchar(char ch, int n) {
for(int j=0; j<n; j++)
    cout << ch;
cout << endl;
}
```

Function Overloading with Different types of Arguments

```
// demonstrates overloaded functions
#include <iostream>
using namespace std;

struct Distance {
   int feet;
   float inches;
};

void dispDist( Distance dd ) {
     cout << dd.feet << “\’-” << dd.inches
<< “\””;
}

void dispDist( float dd ) {
     int feet = static_cast<int>(dd / 12);
     float inches = dd - feet*12;
     cout << feet << “\’-” << inches << “\””;
}
```

```
int main() {
     Distance d1;
     float d2;
     cout << “\nEnter feet: “;
     cin >> d1.feet;
     cout << “Enter inches: “;
     cin >> d1.inches;
     cout << “Enter entire distance
          in inches: “;
     cin >> d2;
     cout << “\nd1 = “;
     dispDist(d1);
     cout << “\nd2 = “;
     dispDist(d2);
     cout << endl;
     return 0;
}
```