In C++ there are two types of functions, *predefined functions* and *user-defined functions*.

- Predefined functions come with libraries of C++.
- User-defined functions are created by programmers.

### Predefined Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Type of Arguments</th>
<th>Type of Value Returned</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pow(x,y)</code></td>
<td>double</td>
<td>double</td>
<td>$x^y$</td>
</tr>
<tr>
<td><code>sqrt(x)</code></td>
<td>double</td>
<td>double</td>
<td>square root of $x$</td>
</tr>
<tr>
<td><code>abs(x)</code></td>
<td>int</td>
<td>int</td>
<td>$</td>
</tr>
<tr>
<td><code>fabs(x)</code></td>
<td>double</td>
<td>double</td>
<td>$</td>
</tr>
<tr>
<td><code>floor(x)</code></td>
<td>double</td>
<td>double</td>
<td>largest integer $\leq x$</td>
</tr>
<tr>
<td><code>ceil(x)</code></td>
<td>double</td>
<td>double</td>
<td>smallest integer $\geq x$</td>
</tr>
</tbody>
</table>
Example

To use predefined functions, the user should know the argument type and the return type of that function. For instance, square root function $\text{sqrt}(x)$ has double as argument type and double as return type. This means if the user wants to find the square root of 16 and assign it to the variable named $\text{Root}$.

Example

The user should declare $\text{Root}$ as type `double` as follows:
```c++
    double Root;
    Root = sqrt(16.0);
```

The expression $\text{sqrt}(16.0)$ is called a function call, and it returns the value of 4.0. This value 4.0 is then assigned to $\text{Root}$.

Function Calls

- One function calls another by using the name of the called function next to ( ) enclosing an argument list.
- A function call temporarily transfers control from the calling function to the called function.

Function Call Syntax

- The argument list is a way for functions to communicate with each other by passing information.
- The argument list can contain 0, 1, or more arguments, separated by commas, depending on the function.
User-Defined Functions

A user-defined function provides us a way to break down a complex program into subprograms. These subprograms are easier to design and understand.

When a function is called

Temporary memory is set up (for its value parameters and any local variables, and also for the function's name if the return type is not void).

Then the flow of control passes to the first statement in the function’s body. The called function's body statements are executed until one of these occurs:

- return statement (with or without a return value), or,
- closing brace of function body.

Then control goes back to where the function was called.

Example

we want to write a function that will determine a letter grade from the given average score.

In general, the average score can be obtained from different methods depend upon the given raw scores, e.g. 3 test scores and a final score or 2 test scores, quiz scores, and a final score.

Example

This implies that we have to write two different programs to assign a letter grade for each problem.

It will be useful if we have a function to determine a letter grade to use regardless of the raw scores.

This means we only have to write a program to compute the average and call the function that determines the letter grade.
#include <iostream>
#include <cmath>

char AssignGrade(int); // Function prototype

int main(void)
{
    int Test1, Test2, Test3, Final;
    int NormalizeScore;
    double Score;
    char FinalGrade;

    cin >> Test1 >> Test2 >> Test3 >> Final;
    Score = (Test1 + Test2 + Test3 + Final) / 4.0;
    NormalizeScore = ceil(Score);
    NormalizeScore = (NormalizeScore / 10) * 10;
    FinalGrade = AssignGrade(NormalizeScore);
    cout << "Your grade is " << FinalGrade;
    return 0;
} //main

char AssignGrade(int Score)
/* Pre-condition: Score is an integer.
Post-condition: The value returned is a letter grade corresponding to
the score. */
{
    char Grade;
    switch (Score)
    {
        case 100:
            Grade = 'A';
            break;
        case 90:
            Grade = 'B';
            break;
        case 80:
            Grade = 'C';
            break;
        case 70:
            Grade = 'D';
            break;
        default:
            Grade = 'F';
    }
    return Grade;
} //AssignGrade

Structure of Functions

There are 2 main steps to create user-defined functions besides main function.

- **Function prototype**
- **Function definition**: Function definition consists of two parts
  - **Function header**
  - **Body of the function**

Function Prototype

- This similar to declaring an identifier.
- This part is to announce that there is/are other function/functions in the program besides main function.
- **Function prototype** comes between header files and main function.
Function Prototype

The structure of function prototype is as follows:

```
returnType functionName (list of formal-parameter types)
```

```
char AssignGrade(int);
double AreaOfRectangle(double, double);
double Power(double Base, int Exponent);
```

Function Definition

```
char AssignGrade(int Score) // Function header
{
    // Pre-condition: Score is an integer.
    // Post-condition: The value returned is a letter grade corresponding to the Score.
    char Grade;
    switch (Score)
    {
        ...
        return Grade;
    }
} // The end of the body of AssignGrade function
```

Example

```
#include <iostream>
#include <cmath>

char AssignGrade(int); // Function prototype

int main(void)
{
    //main
}
```

```
double AreaOfRectangle(double Width, double Length) // Function header
{
    // Pre-condition: Width and Length of the rectangle.
    // Post-condition: The value returned is the area of the rectangle.
    return Width * Length;
} // The end of the body of AreaOfRectangle function
```

```
```
Example

```cpp
double Power(double Base, int Exponent) // Function header
   /* Pre-condition: Base and Exponent. */
   /* Post-condition: The value returned is Base^Exponent. */
   /* Beginning of the body of Power function */
   double Product = 1;
   int AbsExponent, Count;
   if (Exponent > 0)
   
   for (Count = 1; Count <= Exponent; Count++)
      Product = Product * Base;
   else if (Exponent < 0)
   
      for (Count = 1; Count <= abs(Exponent); Count++)
      Product = Product * Base;
   else
      Product = 1;

   return Product;
   /* End of the body of Power function */
```

To Compile Successfully,

Before a function is called in your program, the compiler must previously process either the function’s prototype, or the function’s definition (heading and body)

Function Parameters

- **Actual parameters** are parameters in function call, e.g., `NormalizeScore`.
- **FinalGrade = AssignGrade(NormalizeScore);**
- **Formal parameters** are parameters declared in the function header, e.g., `Score, Width, Length, Base, and Exponent`.

To Compile Successfully,

Before a function is called in your program, the compiler must previously process either the function’s prototype, or the function’s definition (heading and body)

Function Parameters

Note that the return type and the list of formal parameters of a function can be **void**. This means the function returns nothing if the return type is **void**, and there is no formal parameter if the list of formal parameters is **void**.
**void Functions**

- **Example:** `DisplayMessage()` which can be called from `main()` to describe the pollution index value it receives as a parameter.
- Your city describes a pollution Index
  - less than 35 as “Pleasant”,
  - 35 through 60 as “Unpleasant”,
  - and above 60 as “Health Hazard.”

```cpp
Example

```cpp
void DisplayMessage(int index) // Function header
{
    if (index < 35)
        cout << "Pleasant";
    else if (index <= 60)
        cout << "Unpleasant";
    else
        cout << "Health Hazard";
} // The end of the body of function
```

**Rest of the Program**

```cpp
#include <iostream>

void DisplayMessage(int); // Function prototype
using namespace std;

int main()
{
    int pollutionIndex;
    cout << "Enter air pollution index"
        << endl;
    cin >> pollutionIndex;
    DisplayMessage(pollutionIndex); // function call
    return 0;
}
```

**void function does NOT return value**

```cpp
void DisplayMessage (int n)
{
    cout << n << " indian"
        << endl;
}
```
Formal Parameters

There are 2 types of formal parameters:
- Value parameters
- Reference (Variable) parameters

Value Parameters

- A value parameter is a formal parameter that receives a copy of the contents of the corresponding actual parameter.
- This means the value of the actual parameter is independent from the value of the formal parameter.

Example of Value Parameters

For example, a function call

\[
\text{ExpVal} = \text{Power}(\text{MainBase}, \text{Exp});
\]

Where MainBase and Exp are actual parameter. The function definition (heading) is

\[
\text{double Power(double Base, int Exponent)}
\]
Reference (Variable) Parameters

- A Reference parameter is a formal parameter that receives the memory location of the corresponding actual parameter.
- *This means the value of the actual parameter will change if the value of the formal parameter changes.*

Variable Parameters

For example, a function call

```
ExpVal = Power(MainBase, Exp);
```

Where `MainBase` and `Exp` are actual parameter. The function definition (heading) is

```
double Power(double& Base, int Exponent)
```

Base and Exponent are formal parameters.

- When the function call in the main function is executed, a connection for variable parameter `Base` to memory locations of `MainBase` is established. The temporary memory location for `Exponent` is also established and a copy of the content of `Exp` is made as content `Exponent`. 
Example

```cpp
void FindAverage(double& Average) // Function header
{" Pre-condition: None.
Post-condition: The value returned is an average of scores. */
}////Beginning of the body of FindAverage function
int Test1, Test2, Test3, Final;
cout >> "Enter three test scores and a final score";
cin >> Test1 >> Test2 >> Test3 >> Final;
Average = (Test1 + Test2 + Test3 + Final) / 4.0;
} //The end of the body of FindAverage function
```

Caution

One must be very careful to make a function call when one or more formal parameters of the function to be called are reference formal parameters.

Caution

The actual parameters in the function call must be identifier (variable) names. They cannot be expression. For example, the function `Power`

```cpp
double Power(double& Base, int Exponent)
```

A function call

```cpp
    ExpVal = Power(MainBase, Exp);
```

is a correct call. But

```cpp
    ExpVal = Power(2.0, Exp);
```

will produce compilation error.
Value Parameters

For value formal parameters, the actual parameters can be any type of expressions since their contents do not change. For example, the function `Power`:

```cpp
double Power(double Base, int Exponent)
```

The followings are valid function calls:

- `ExpVal = Power(MainBase, Exp);`
- `ExpVal = Power(2.0, 5);`
- `ExpVal = Power(2.0*MainBase, 5);`
- `ExpVal = Power(2.0*3, Exp);`
- `ExpVal = Power(MainBase, 4*Exp);`

Type of Variables

- **Local variables:** A local variable is an identifier declared in a function. Local variables can only be accessed in side its own function.
- **Global variables:** A global variable is an identifier declared outside of any functions. Global variables can be accessed by any functions.

Example

```cpp
#include <iostream>
#include <cmath>

char AssignGrade(int); // Function prototype

int main (void)
{
    int Test1, Test2, Test3, Final;
    int NormalizeScore;
    double Score;
    char FinalGrade;

    return 0;
} //main
```
char AssignGrade (int Score)
/* Pre-condition: Score is an integer.
Post-condition: The value returned
is a letter grade corresponding to 
the Score. */
{
    char Grade;
    return Grade;
} //AssignGrade

Test1, Test2, Test3, Final,
NormalizeScore, Score, FinalGrade are local 
variables of function main.
Grade is a local variable of function AssignGrade.

Preconditions and Postconditions

- The precondition is an assertion describing everything that the function requires to be true at the moment the function is invoked.
- The postcondition describes the state at the moment the function finishes executing.
- The caller is responsible for ensuring the precondition, and the function code must ensure the postcondition.

Function with Postconditions

void GetRating (/* out */ char& Letter)
//Precondition: None
//Postcondition: User has been prompted to enter a character
//   && Letter == one of these input values: E,G,A, or P
{
    cout << "Enter employee rating." << endl;
    cout << "Use E, G, A, or P : " ;
    cin >> letter;
    while( (letter != 'E') && (letter != 'G') 
      && (letter != 'A') && (letter != 'P') )
    {
        cout << "Rating invalid. Enter again : " ;
        cin >> letter;
    }
}
Function with Preconditions and Postconditions

```c
void GetRoots( /*in*/ float a, /*in*/ float b, /*in*/ float c,
              /*out*/ float& root1, /*out*/ float& root2)
//  Precondition:   a, b, and c are assigned,
// &&   a != 0    &&   b*b - 4*a*c != 0
//  Postcondition:    root1 and root2 are assigned
//   &&  root1 and root2 are roots of quadratic with coefficients a, b, c
{
    float temp;
    temp = b * b - 4.0 * a * c;
    root1 = (-b + sqrt(temp) ) / ( 2.0 * a );
    root2 = (-b - sqrt(temp) ) / ( 2.0 * a );
}
```

Function with Preconditions and Postconditions

```c
void Swap( /*inout*/ int& firstInt, /*inout*/ int& secondInt)
//Precondition:  firstInt and secondInt are assigned
//Postcondition: firstInt == secondInt@entry
//   && secondInt == firstInt@entry
{
    int temporaryInt;
    temporaryInt = firstInt;
    firstInt = secondInt;
    secondInt = temporaryInt;
}
```