

Functions

C++ Notes

Mrs. Alano

Functions: modules that perform a group of tasks

- Also known as a **subroutine**, **procedure**, or **method**

In C++, every function has a **header**, or introductory line, that includes three parts:

- The **function's return type**
- The **name** or identifier of the function
- In parentheses, the types and names of any variables that are passed to the function

```
void displayLogo()  
{  
}  
}
```

A function is like a black box

```
int main()  
{  
    cout<<"Hello from:"<<endl;  
    displayLogo();  
}
```

← Function call



When you write a main() function, you can place any other function (**subfunction**) that is used by main() in one of three different locations:

- As part of the same file, before the main() function
- **As part of the same file, after the main() function (This is the method we will use)**
- In its own file

```

#include<iostream>
#include<string>

using namespace std;

int main()
{
    string lastName, salesRep;
    void formLetter();
    cout<<"Enter name of family ";
    cin>>lastName;
    cout<<"Enter name of sales representative ";
    cin>>salesRep;
    cout<<endl<<"Dear "<<lastName<<" Family:"<<endl;
    formLetter();
    cout<<"Sincerely,"<<endl<<salesRep<<endl;
}

void formLetter()
{
    cout<<"  Thank you for your interest in our vacation resort. "<<endl<<
    "Enclosed is a color brochure featuring our beautiful time-"<<endl<<
    "share units. I will be in contact in a few days to see "<<endl<<
    "whether you have any questions."<<endl;
}

```

Benefits of writing functions:

- Functions can be written once and subsequently included in any number of programs
- To change a function, you make the change in one location and it is automatically applied in all the programs that use the function
- Within a program's main() function, you can con-dense many related actions into a single function call
- When you use functions that have already been written and tested, you gain an additional benefit: you do not worry about how it works; you just call it

Reusing software components effectively facilitates **maintenance programming** and is a hallmark of object-oriented programming

```

int main()
{
    getEmployeeData();
    computeGrossPay();
    computeAndDeductFederalTaxes();
    computeAndDeductLocalTaxes();
    deductInsurancePremium();
    printPaycheck();
}

```

Scope of Variables

- Some variables can be accessed throughout an entire program, while others can be accessed only in a limited part of the program
- The **scope** of a variable defines where it can be accessed in a program
- To avoid conflicts between local and global variables with the same name, you use the scope resolution operator
- **Global** variables are known to all functions
- **Local** variables are known only in a limited scope
 - Created when they are declared within a block
 - Known only to that block
 - Cease to exist when their block ends
- **To override** means to take precedence over

```
#include<iostream>
using namespace std;
int main()
{
    int b = 2; // b comes into existence
    cout<<b;
    {
        int c = 3; // c comes into existence
        cout<<c;
    } // c goes out of scope – you can't use c anymore
    cout<<b;
} // b goes out of scope – you can't use b anymore
```

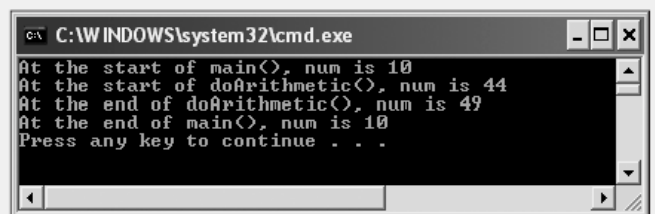
- A variable is **in scope (accessible)** between the time it is declared and when its block of code ends; usually this is when a variable appears between braces in a program
- After the ending brace where a variable was declared, the variable is **out of scope** (inaccessible)
- The **lifetime** of a variable is the time during which it is defined—from declaration to destruction

```
#include<iostream>
#include<string>

using namespace std;

int main()
{
    void doArithmetic();
    int num = 10;
    cout<<"At the start of main(), num is "<<num<<endl;
    doArithmetic();
    cout<<"At the end of main(), num is "<<num<<endl;
}

void doArithmetic()
{
    int num = 44;
    cout<<"At the start of doArithmetic(), num is "<<num<<endl;
    num = num + 5;
    cout<<"At the end of doArithmetic(), num is "<<num<<endl;
}
```



```
C:\WINDOWS\system32\cmd.exe
At the start of main(), num is 10
At the start of doArithmetic(), num is 44
At the end of doArithmetic(), num is 49
At the end of main(), num is 10
Press any key to continue . . .
```

- **Encapsulation**, or data hiding, is the principle of containing variables locally in their functions
- **Disadvantages of using global variables over local variables:**
 - If variables are global in a file and you reuse any functions in a new program, the variables must be redeclared in the new program
 - Global variables can be altered by any function, leading to errors

Returning Values from functions

A **return** value is a value sent from a subfunction to the function that called it

```
#include<iostream>
using namespace std;
// main program
int main()
{
    char usersInitial;
    char askUserForInitial();
    usersInitial = askUserForInitial();
    cout<<"Your initial is "<<usersInitial<<endl;
}
// subfunction
char askUserForInitial()
{
    char letter;
    cout<<"Please type your initial and press Enter "<<endl;
    cin>>letter;
    return letter;
}
```

Passing Values to Functions

- Functions get data from other functions by accepting **parameters** from the sending function
 - The value passed is called an **argument**
- You can write the definition, or prototype, for the `computeTax()` function that requires a double parameter and returns nothing in one of two ways:
 - `void computeTax(double);`
 - `void computeTax(double price);` //provides documentation

```
#include<iostream>
using namespace std;
int main()
{
    double price;
    void computeTax(double);
    cout<<"Enter an item's price ";
    cin>>price;
    computeTax(price);
}

void computeTax(double amount)
{
    double tax;
    const double TAX_RATE = 0.07;
    tax = amount * TAX_RATE;
    cout<<"The tax on "<<amount<<" is "<<tax<<endl;
}
```

Function prototype/declaration

Function header

Passing variables by Value – sends a copy of the value in the variable to the function

Output: Dividend is 2 and modulus is 5

```
#include<iostream>
using namespace std;
int main()
{
    int a = 19, b = 7, dividend, modulus;
    int getDiv(int, int);
    int getRemainder(int, int);
    dividend = getDiv(a, b);
    modulus = getRemainder(a, b);
    cout<<"Dividend is "<<dividend<<
        " and modulus is "<<modulus<<endl;
}
int getDiv(int num1, int num2)
{
    int result;
    result = num1 / num2;
    return result;
}
int getRemainder(int num1, int num2)
{
    int result;
    result = num1 % num2;
    return result;
}
```

Passing Variable Addresses to Reference Variables – sends the memory address of the variable to the function – so changes made in the function show up in main

Output: Dividend is 2 and modulus is 5

```
#include<iostream>
using namespace std;
int main()
{
    int a = 19, b = 7, dividend, modulus;
    void getResults(int, int, int&, int&);
    getResults(a, b, dividend, modulus);
    cout<<"Dividend is "<<dividend<<" and modulus is "<<
        modulus<<endl;
}
void getResults(int num1, int num2, int &oneAddress,
    int &anotherAddress)
{
    oneAddress = num1 / num2;
    anotherAddress = num1 % num2;
}
```

Using Objects as Arguments to Functions and as Return Types of Functions

```
int main()
{
    Customer oneCustomer;
    Customer getCustomerData();
    void displayCustomerData(Customer);

    oneCustomer = getCustomerData();
    displayCustomerData(oneCustomer);
}
Customer getCustomerData()
{
    Customer cust;
    cout<<"Enter customer ID number ";
    cin>>cust.custId;
    cout<<"Enter customer's last name ";
    cin>>cust.custName;
    return cust;
}
void displayCustomerData(Customer customer)
{
    cout<<"ID is #"<<customer.custId<<
        " and name is "<<customer.custName<<endl;
}
```

Note - In C++ Objects are passed by Value – meaning a copy of the values in the object are sent to the subfunction.

Passing Arrays to Functions

```
#include<iostream>
using namespace std;
int main()
{
    const int SIZE = 4;
    int nums[SIZE] = {4, 21, 300, 612};
    void increaseArray(int[], const int);
    int x;
    for(x = 0; x < SIZE; ++x)
        cout<<nums[x]<<" ";
    cout<<endl;
    increaseArray(nums, SIZE);
    for(x = 0; x < SIZE; ++x)
        cout<<nums[x]<<" ";
    cout<<endl;
}
void increaseArray(int values[], const int SZ)
{
    int y;
    for(y = 0; y < SZ; ++y)
        ++values[y];
}
```

Note - In C++ arrays are passed by reference – meaning if you change the array in the subfunction it also changes in the main function