Chapter 5
Functions

Learning Objectives

- Use of functions in C++
- Simple functions
- Passing arguments to functions
- Create function overloads
- Recursion
- Write and use inline functions
- Default Arguments
- Variables and Storage Class

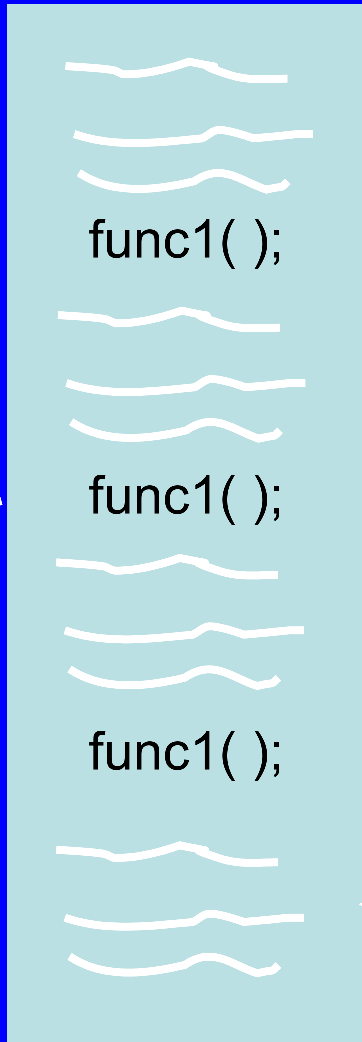
FUNCTIONS

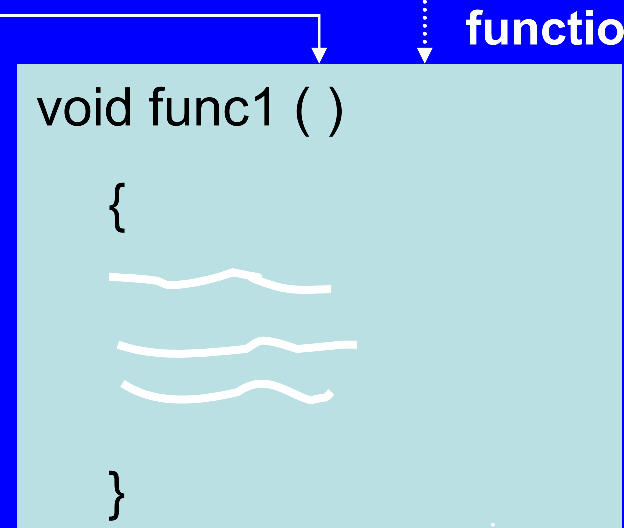
- A function groups a number of program statements into a unit and gives it a name
- This unit can be invoked from other parts of the program

Use of Function

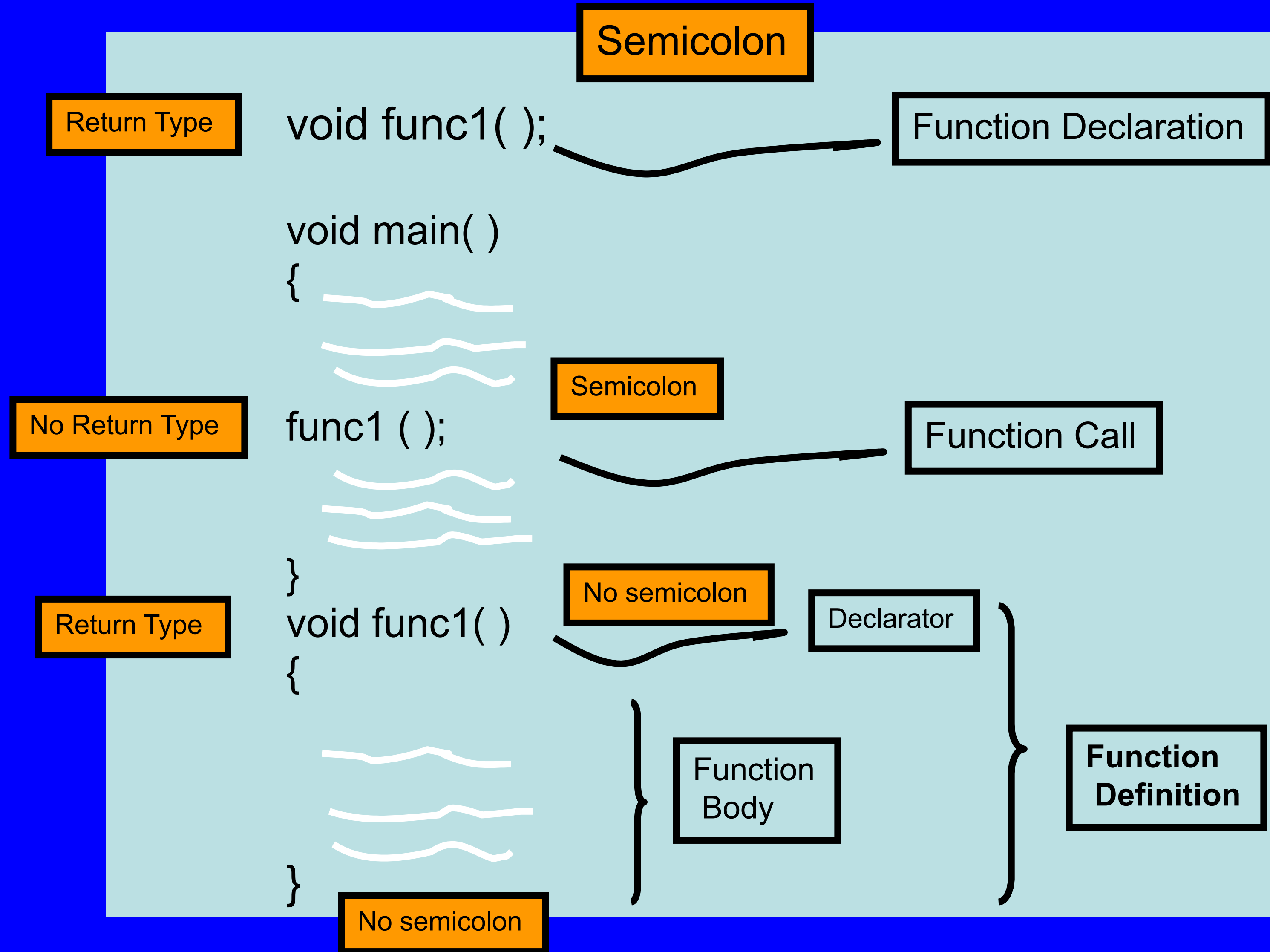
- To aid in the conceptual organization of a program
- To reduce program size

FUNCTIONS





Simple Functions



Function Definition

It contains the actual code of function

```
int ftoc (int temp)
{
        int result= (temp-32)*5/9;
        return result;
}
```

<u>Function Definition Format</u>

```
returntype  functionname (parameterlist)
  {
        Statements;
  }
```

parameterlist contains type and name of variables used in the function

FUNCTION EXAMPLE

```
// table.cpp
// demonstrates simple function
#include <iostream.h>

void starline();                    //function declaration

int main()
{
    starline();                                         //call to function
    cout << "Data type  Range" << endl;
    starline();                                         //call to function
    cout << "char   -128 to 127" << endl
         << "short  -32,768 to 32,767" << endl
         << "int      System dependent" << endl
         << "long    -2, 147, 483, 648 to 2,147, 483, 647" << endl;
    starline();                                         //call to function
return 0;
}
```

Cont…..

FUNCTION EXAMPLE

```
//…………………………………………
//starline()
//function definition
    void starline()          //function declarator
    {
        for(int j=0; j<45; j++)  //function body
        cout << '*';
        cout << endl;
    }
```

Simple Functions Contd..

- Comparison with Library Functions
  - The declaration is in the header file specified at the beginning of the program
  - The definition (compiled into executable code) is in a library file that is linked automatically
- Eliminating the Declaration
  - Place the definition (the function itself) in the beginning of program before the first call to the function

Eliminating the Declaration Example

```
// table2.cpp
// demonstrates function definition preceding function calls
#include <iostream.h>
                                  //no function declaration
//...............................
// starline()                      //function definition
void starline()
{
for(int j=0; j<45; j++)
cout << '*';
cout << endl;
}
//............................................
int main()                        //main() follows function
{
starline();                       //call to function
cout << "Date type Range" << endl;
starline();                       //call to function
cout << "char                     -128 to 127" << endl
       << "short                  -32, 768 to 32, 767" << endl
       << "int                    System dependent" << endl
       << "long                   -2, 147, 483, 648 to 2, 147, 483, 647" << endl;
starline();                       //call to function
return 0;
}
```

Passing Arguments to Function

- Constants
- Expressions
- Variables
  - By value
  - By reference

Passing Arguments to Functions Contd..

- The parameter of function are initialized to the value of argument passed to it
- Function create copies of the arguments passed to it, is called passing by value
- Passing other types of arguments
  - Pointers and arrays
  - Structures
  - Objects

Passing Constant Example

```
// tablearg.cpp
// demonstrates function arguments
#include <iostream.h>

void repchar (char, int);
    //function declaration
int main( )
{
repchar('-' , 43);                    //call to
    function
cout << "Date type   Range" << endl;
repchar('=', 23);                     // call to
    function
cout << "char   -128 to 127" << endl
     << "short   -32, 768 to 32, 767" << endl
     << "int system dependent" << endl
    << "double  -2, 147, 483, 648 to 2, 147, 483,
    647" << endl;
repchar('-', 43);                     // call to
    function
return 0;
}
```

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

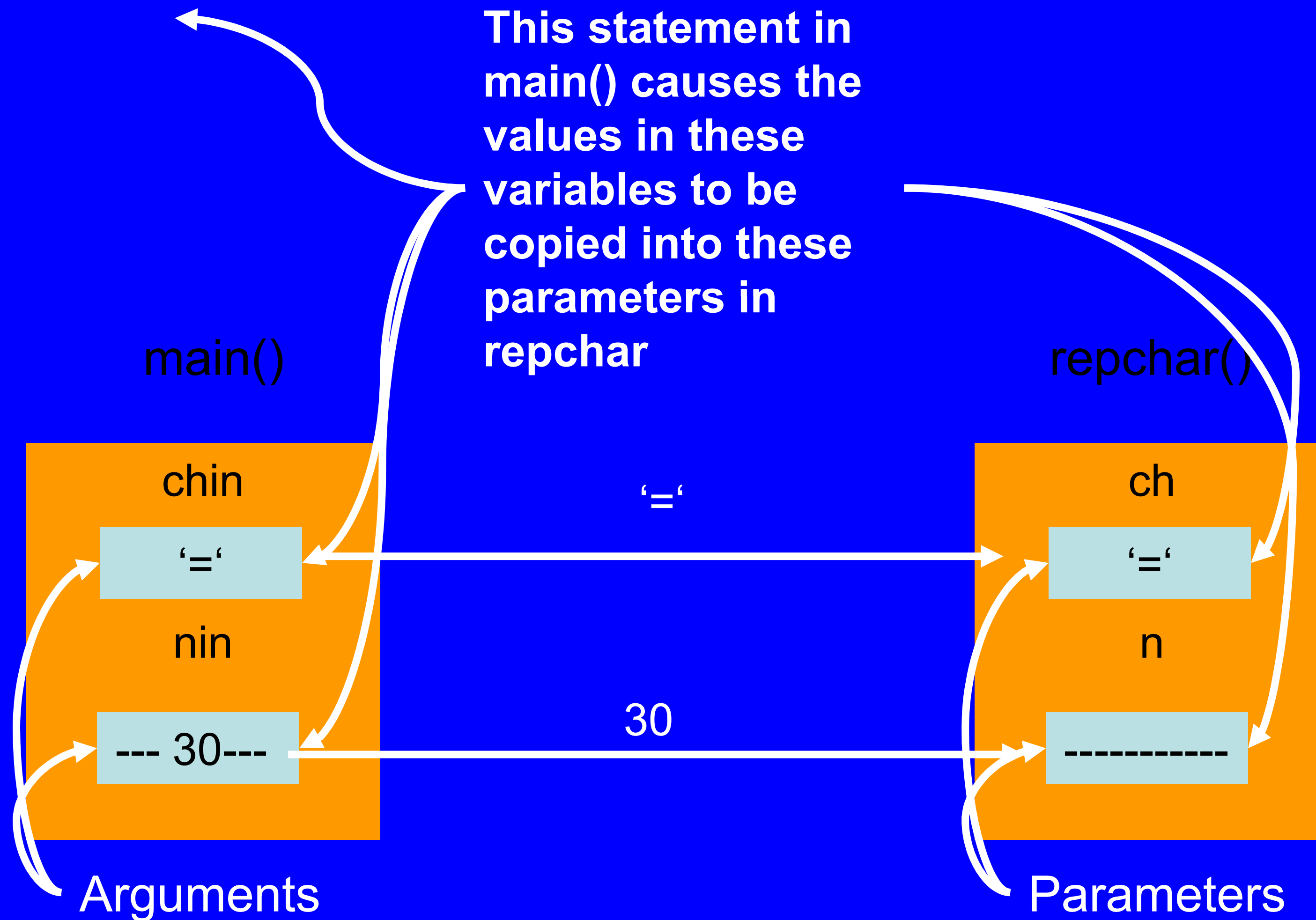
Passing Variable Example

```
// vararg.cpp
// demonstrates variable arguments
#include <iostream>
using namespace std;
void repchar(char, int);
int main()
{
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;
}
```

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

Variable Passed by Value

Repchar (chin,nin);



Example: Passing Variables

```
#include <iostream.h>

int ftoc (int);        //function declaration

int main()
{
    int ftemp;
    cout << "Enter temperature in degree Fahrenhite: ";
        cin >> ftemp;
        int ctemp = ftoc(ftemp);
        cout << "The temperature " <<ftemp<< " Degree Fahrenhite equals
    "
                <<ctemp<< " Degree Centigrade "<<endl;
return 0;
}
//..............................................
//function definition
    int ftoc (int temp)
    {
        return  (temp-32)*5/9;
    }
```

Example: Passing constant, Variable & Expression

```
#include <iostream.h>

int square(int);

int main()
{
    int x = 5, y= 7;
    cout << square(9) <<endl
         << square (x) << endl
         <<square (x+y) << endl
         << square (x+y+7) <<endl;

return 0;
}

int square(int a)

{
    return a *=a;
}
```

Returning Values from Function

- Return value consist of an answer to the problem the function has solved
- When a function returns a value, the data type of this value must be specified
- The function declaration does this by placing the data type e.g.

```
float lbstokg(float);
```

before the function name in the declaration and the definition

Returning Values from Function

```
// convert.cpp
// demonstrates return values, converts pounds to kg
#include <iostream.h>

float lbstokg(float);      //declaration

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

Returning Values from Function

```
//...............................
// lbstokg()
// converts pounds to kilograms
float lbstokg(float pounds)
{
float kilograms = 0.453592 * pounds;
return kilograms;
}
```

The Return Statement

- The function lbstokg() is passed an argument representing a weight in pounds, which it stores in the parameter pounds.
- The result is stored in the variable kilogram
- The value then returned to the calling program using a return statement

```
return kilogram;
```

Return Statement

kgs = lbstokg(lbs);

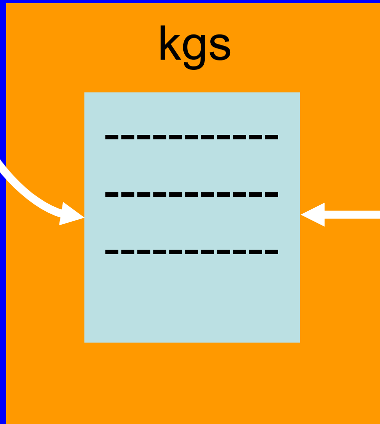
return kilograms;

2. This statement in main() causes the Return value to be assigned to this variable

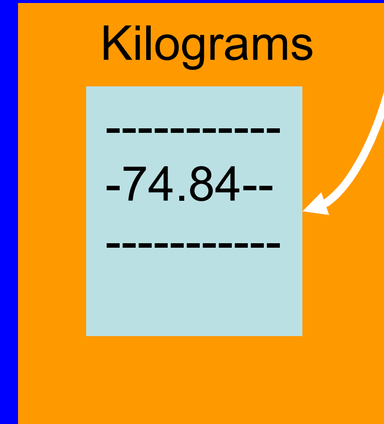
1.This statement in lbstokg() causes the return value to be assigned to this variable

main()

repchar()



74.84



Reference Arguments

- A reference provides an alias – a different name – for a variable
- Passing arguments by value is useful when the function does not to modify the original variable in the calling program.
- It offers insurance that the function cannot harm the original variable
- Passing arguments by reference uses a different mechanism. Instead of a value being passed to the function , a reference to the original variable , in the calling program is passed

Reference Arguments

- An important advantage of passing by reference is that the function can access the actual variable 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

Referencing: Creating Aliases

```
#include <iostream.h>

int main()
{
    int x = 3;
    int &y = x;
    cout<<"x= "<<x<<endl<<"y= "<<y<<endl;
    y=7;
    cout<<"x= "<<x<<endl<<"y= "<<y<<endl;
    x=33;
    cout<<"x= "<<x<<endl<<"y= "<<y<<endl;
return 0;
}
```

Referencing : Example

```
// comparing pass by value and pass by reference
#include <iostream>
using namespace std;

int squareByValue(int);
void squareByReference(int&);

int main()
{
    int x = 2, z = 4;
    cout << "x = "<<x<< " before squareByValue\n"
            <<"Value returned by squareByValue: "
            << squareByValue(x)<<endl
    <<"x= "<<x<< " after squareByValue\n"<<endl;


    cout <<"z = "<<z<< " before
    squreByReference"<< endl;
    squareByReference(z);
    cout << "z = " << z << " after
    squareByReference" << endl;

return 0;
}
```

```
int squareByValue(int a)

{
    return a *=a;
}

void squareByReference (int
    &y)
{
    y *=y;
}
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x= 2 after squareByValue

z = 4 before squreByReference
z = 16 after squareByReference
```

Passing Simple Data types by Reference

```
// ref.cpp
// 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: ";                  //number from user
cin >> number;
intfrac(number, intpart, fracpart);                 //find int and frac
cout << "Integer part is " << intpart               //print them
      << " fraction part is  " << fracpart << endl;
}  while ( number != 0.0 );                         //exit loop on 0.0
return 0;
}
```

Passing Simple Data types by Reference

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

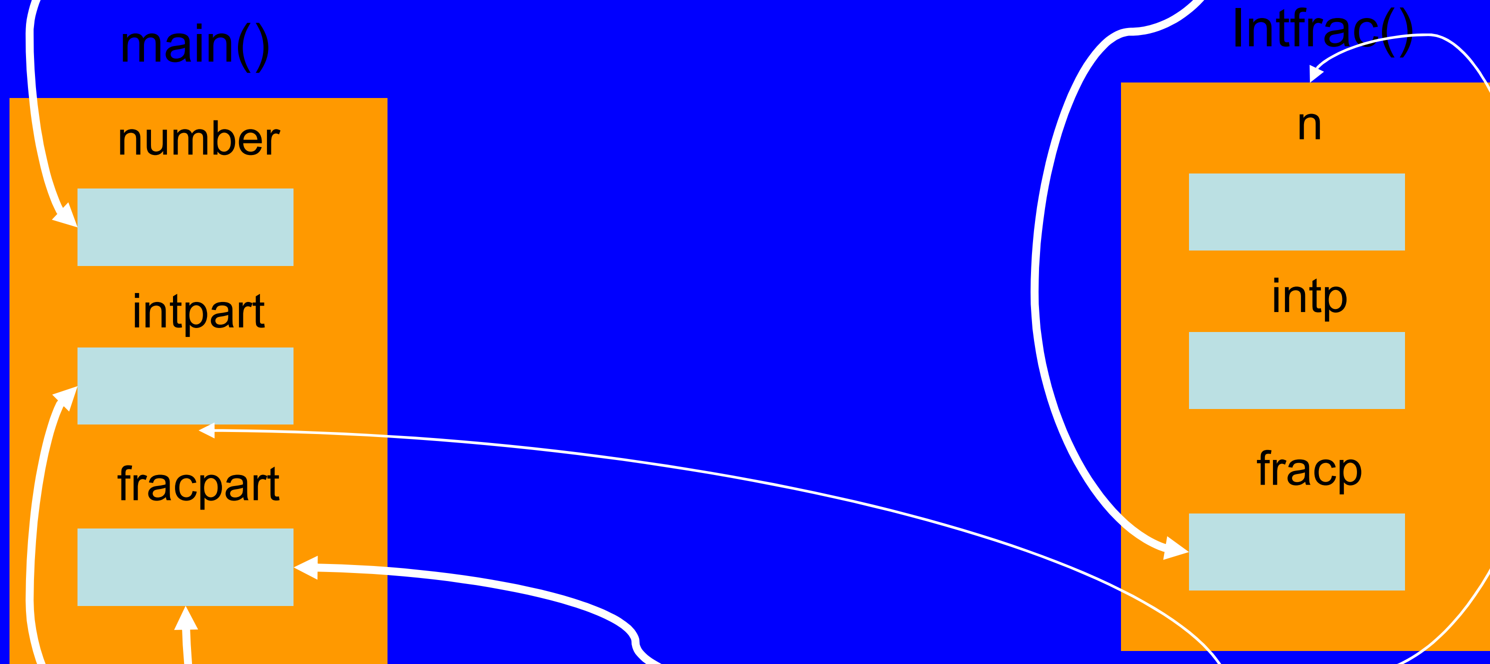
Passing Simple Data types by Reference

- The program will separate this number into an integer and a fractional part i.e the user number is 12.456 the program should report that the integer part is 12.0 and the fractional part is 0.456
- Library function fmod() performs the similar tasks for type double.

Passing Simple Data types by Reference

Intfrac(number,intpart,fracpart);

This statement in main() causes this variable to be copied into this parameter it also sets up aliases for these variables with these names




```
Fracp=n-intp;
Long temp = static_cast<long>(n);
```

These statements in intfrac() operate on these variable as if they were in intfrac()

A More Complex Pass by Reference

```
// reforder.cpp
// orders two arguments passed by reference
#include  <iostream>
using namespace std;
int main()
{
void order(int&, int&);                    //prototype
int n1=99, n2=11;                          //this pair not ordered
int n3=22, n4=88;                          //this pair ordered
order(n1, n2);                             //order each pair of numbers
order(n3, n4);
cout << "n1=" << n1 << endl;               //print out all numbers
cout << "n2=" << n2 << endl;
cout << "n3=" << n3 << endl;
cout << "n4=" << n4 << endl;
return 0;
}
```

A More Complex Pass by Reference

```
//……………………………….
void order(int& numb1, int& numb2)   //if 1st larger than 2nd,
{
if (numb1>numb2)
{
int temp = numb1;                    //swap them
numb1 = numb2;
numb2 = temp;
}
}
```

Function Overloading

- A C++ programmer may use the same name for more than one function.
- This would typically be done for different but related functions that perform similar tasks.
- *Function overloading* defines a new meaning for a function name that is already in use.

Function Overloading Contd..

- These two functions are different, even though they have the same name

```
int sqrt (int x)
 {
   ....
 }

 double sqrt (double x)
 {
   ....
 }
```

Function Overloading Contd..

- In C++, when the compiler sees a function call, it selects one from its list of known functions by examining both the called function name and the arguments provided in the call.
- When an overloaded function is called the compiler selects the proper function by examining the *number*, *types* and *order* of the arguments in the call.

Function Overloading Contd..

- Function Signature
  - **The combination of a function's name and its parameter list is called its *signature*. Every function must have a unique signature**
  - **Overloaded functions (functions with same name) can have different return types but must have different parameter list.**

Examples : Function Overloading

```
// some function prototypes
   void fun (void);
   int fun (int);                 // valid overloading
   double fun (int, int)          // valid overloading
   double fun (int);              // illegal redefinition
   int fun (void);                // illegal redefinition
```

Function Overloading Example

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

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

int main()
{
repchar();
repchar( '=' );
repchar( '+', 30);
return 0;
}
```

Function Overloading Example

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

     for(int  j=0;  j<n;  j++)  // loops n times
     cout << ch;                //prints specified character
cout << endl;
}
```

RECURSION

- Existence of functions makes possible a programming technique called recursion
- Recursion involves a function calling itself
- Recursion is much easier to understand with an example than with lengthy explanations

RECURSION EXAMPLE

```
//factor2.cpp
//calculate factorials using recursion
#include<iostream>
using namespace std;
unsigned long factfunc(unsigned long);          //declaration
int main()
{
    int n;
unsigned long fact;                //factorial
cout<<"Enter an integer : ";
cin>>n;
fact=factfunc(n);
cout<<"Factorial of "   <<n <<" is " <<fact <<endl;
return 0;
}
```

RECURSION EXAMPLE

```
//.........................
//factfunc()
//calls itself to calculate factorial
unsigned long factfunc(unsigned long n)
{
if (n>1)
return n*factfunc(n-1);     //self call
else
return 1;
}
```

Inline Functions

- To save execution time in short functions
- At a function call in the source file the actual code is inserted, instead of jump to the function
- Inline library functions are usually defined (not just declared) in header files

INLINE FUNCTION EXAMPLE

```
//inline.cpp
//demonstrate inline function
#include<iostream>
using namespace std;
//lbstokg()
//converts pound to kilograms
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 kilogram is " <<lbstokg(lbs)<<endl;
return 0;
}
```

Default Argument

- A function can be called without specifying all its argument
- Default argument is useful if you don’t want to go to the trouble of writing arguments i.e. almost always have the same value
- Using default arguments means that the existing function calls can continue to use the old number of arguments while new functions calls can use more

Default Argument Example

```
//missarg.cpp
//demonstrate missing and default argument
#include<iostream>
using namespace std;
void repchar(char='*',int=45);
int main()
{
repchar();
repchar('=');
repchar('+',30);
return 0;
}
```

```
//..........................
//repcahr()
//display line of characters
void repchar(char ch, int n)
{
for (int j=0;j<n ; j++)
cout <<ch;
cout <<endl;
}
```

Variables and Storage Classes

- Storage Class of a variable
  - It determines which part of the program can access it and how long it stays in existence
- Three storage classes
  - Automatic or Local Variables
  - External or Global Variables
  - Static Local Variables

Automatic Variables or Local Variables

- Variables defined within a function body
- Sometime auto keyword is used
- Lifetime
  - Variable is not created until the function in which it is defined is called
  - When called function returns and control passed to the calling program, the variables are destroyed and their value is lost
- Visibility or Scope
  - Location within a program from which it can be accessed (With in a function)
- Initialization
  - When variable is created it is not initialized by compiler

External or Global Variables

- Variables defined outside a function body
- Lifetime
  - Exists for the life of the program
- Visibility or Scope
  - Visible to all the functions that follow the definition in a program
- Initialization
  - When variable is created it is automatically initialized to 0

Example : Global Variables

```
// extern.cpp
// demonstrates global variables
#include<iostream>
#include<conio.h>
using namespace std;
char ch = 'a';    //global variable
void getachar();
void putachar();
int main()
{
    while(ch != '\r')
    {
        getachar();
        putachar();
    }
    cout<<endl;
    return 0;
}
```

```
//.............
void getachar()
{
    ch=getch();
}
//.............
void putachar()
{
    cout<<ch;
}
```

Static Local Variables

- Lifetime
  - Created on first call of the function and exists for the life of the program
- Visibility or Scope
  - Visible within the function
- Initialization
  - Initialized only once per program

Example : Static Local Variable

```
// static.cpp
// Demonstrates static variables
#include<iostream>
using namespace std;
float getavg(float);

int main()
  {
  float data=1, avg;

  while( data != 0 )
    {
    cout << "Enter a number: ";
    cin >> data;
    avg = getavg(data);
    cout << "New average is "
    << avg << endl;
    }
  return 0;
  }
```

```
//------------------------------------------------------------
// getavg()
// finds average of old plus new data
float getavg(float newdata)
  {
  static float total = 0;  //static variables are initialized
  static int count = 0;    // only once per program
  count++;                 //increment count
  total += newdata;        //add new data to total
  return total / count;    //return the new average
  }
```

Variable Storage Classes Another Example

```
#include<iostream>
using namespace std;

void a(void);          //function declarations
void b(void);
void c(void);
int x = 1;  //global variable
int main()
{
    int x = 5;          //local variable to main

    cout << "local x in outer scope of main is "<<x<<endl;
    {       // starting a new scope
            int x = 7;
            cout << "local x is in inner scope of main is "<<x<<endl;
    }       //end of inner or new scope
    cout << "local x in outer scope of main is "<<x<<endl;
    a();    //func a has automatic local x
    b();    //func b has static local x
    c();    //func c uses a global x
    a();    //func a reinitializes automatic local x
    b();    //static local x retains its previous value
    c();    //global x also retains its value
    cout << "local x in main is "<<x<<endl;
    return 0;
}
```

Contd’

Variable Storage Classes Another Example..Contd

```
void a(void)
{
    int x = 25;      //initialized each time a is called
    cout<<endl<<"local x in a is "<<x<<" after entering a"<<endl;
    ++x;
    cout<<"local x in a is "<<x<<" before exiting a"<<endl;
}
void b(void)
{
    static int x = 50;         //intialization only first time
                                                   //b is called
    cout<<endl<<"local staic x is "<<x<<" on entering b"<<endl;
    ++x;
    cout<<"local static x is "<<x<<" on exiting b"<<endl;
}
void c(void)
{
    cout<<endl<<"global x is "<<x<<" on entering c"<<endl;
    x *= 10;
    cout <<"global x is "<<x<<" on exiting c"<<endl;
}
```

THE END