Chapter 4

Procedural Abstraction and Functions That Return a Value

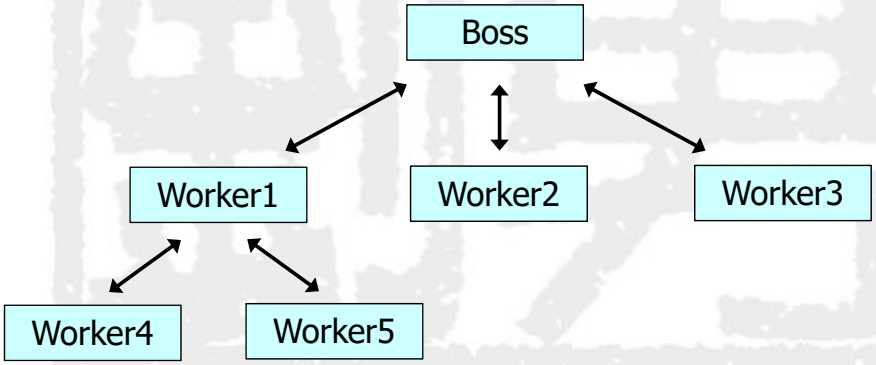
Prof. Chien-Nan (Jimmy) Liu
Dept. of Electrical Engineering
National Chiao-Tung Univ.

Tel: (03)5712121 ext:31211
E-mail: jimmyliu@nctu.edu.tw
http://mseda.ee.nctu.edu.tw/jimmyliu

Overview

Top-Down Design

- Top Down Design (also called stepwise refinement)
  - Break the algorithm into subtasks
  - Break each subtask into smaller subtasks
  - Eventually the smaller subtasks are trivial to implement in the programming language



Benefits of Top-Down Design

- Subtasks, or functions in C++, make programs
  - Easier to understand
  - Easier to write & change
  - Easier to test & debug
  - Easier for teams to develop
- Motivations for "functionalizing" a program
  - Divide-and-conquer makes program development more manageable
  - Software reusability—using existing functions as building blocks to create new programs
  - Avoid repeating code in a program
    - Packaging code as a function allows the code to be executed from different locations in a program

Overview

Predefined Functions

- C++ comes with libraries of predefined functions
- Example: sqrt function (in cmath library)
  - theRoot = sqrt(9.0); //compute the square root of a number
  - The number, 9, is called the argument
  - theRoot will contain 3.0
- sqrt(9.0) is a function call
  - Stop the execution and jump into the sqrt function
  - After finished, it returns the result and resume execution
  - The argument (9) can also be a variable or an expression
- A function call can be used like any expression
  - bonus = sqrt(sales) / 10;
  - Cout << "The side of a square with area " << area
    << " is " << sqrt(area);

Example of Function Call

```
//DISPLAY 4.1 A Function Call
//Computes the size of a dog house that can
be purchased given the user's budget.
#include <iostream>
#include <cmath>
using namespace std;

int main( )
{
    const double COST_PER_SQ_FT = 10.50;
    double budget, area, length_side;

    cout << "Enter the amount budgeted for
              your dog house $";
    cin >> budget;

    area = budget/COST_PER_SQ_FT;

    length_side = sqrt(area);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "For a price of $" << budget << endl
         << "I can build you a luxurious square dog
              house\n"
         << "that is " << length_side
         << " feet on each side.\n";

    return 0;
}
```

Function Call Syntax

- Function_name (*Argument_List*)
  - *Argument_List* is a comma separated list: (Argument_1, Argument_2, … , Argument_Last)
- Example:
  - side = sqrt(area);
  - cout << "2.5 to the power 3.0 is " << pow(2.5, 3.0);
- The corresponding library must be "included" in a program to make the predefined functions available
- To include the math library containing sqrt():
  #include <cmath>

Some Common Math Functions

Some Predefined Functions

Name	Description	Type of Arguments	Type of Value Returned	Example	Value	Library Header
sqrt	square root	*double*	*double*	sqrt(4.0)	2.0	cmath
pow	powers	*double*	*double*	pow(2.0,3.0)	8.0	cmath
abs	absolute value for *int*	*int*	*int*	abs(-7) abs(7)	7 7	cstdlib
labs	absolute value for *long*	*long*	*long*	labs(-70000) labs(70000)	70000 70000	cstdlib
fabs	absolute value for *double*	*double*	*double*	fabs(-7.5) fabs(7.5)	7.5 7.5	cmath
ceil	ceiling (round up)	*double*	*double*	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	floor (round down)	*double*	*double*	floor(3.2) floor(3.9)	3.0 3.0	cmath

Random Number Generation

- The *rand()* function returns a random integer between 0 and RAND_MAX (a large constant defined in library)

```
#include <cstdlib>
i = rand();
```

- *rand()* actually generates pseudo-random numbers
  - The numbers in the sequence appear to be random, but the sequence repeats itself each time the program executes
  - Ex: First execution → 6 1 4 6 2 1 6 1 6 4 …

    Second execution → 6 1 4 6 2 1 6 1 6 4 …
- Function *srand* takes an unsigned integer argument and seeds the rand function
  - Changing different starting point will produce different sequence of random numbers
  - Ex: srand(67); or srand(num);

Using Random Numbers

- How to generate random numbers in a specific range?
- Ex: Rolling a six-sided die (a random number from 1 to 6)

  → `int die = (rand() % 6) + 1;`
  - Scale: rand() % 6 → the remainder is from 0 to 5
  - Shift: (rand() % 6) +1 → (0 to 5) becomes (1 to 6)
- Generate a random number x where 10 < x < 21

  → `int x = (rand() % 10) + 11;` → Actual range is 11 to 20, total 10 numbers
- Be aware of probability distribution (normal or uniform?)
  - random1 (1~6) + random2 (1~6) ≠ random (2~12)
- To change the random seed automatically, use time+srand

```
#include <ctime>
srand(time(0));
```

→ time(0) returns the current time as the number of seconds since January 1, 1970, at midnight Greenwich Mean Time (GMT).

Type Casting

- Recall the problem with integer division:

```
int totalCandy = 9, numberOfPeople = 4;
double candyPerPerson = totalCandy / numberOfPeople;
```

  - candyPerPerson = 2, not 2.25!
- A Type Cast produces a temporary value of another type
  - **static_cast<double>**(totalCandy) / numberOfPeople
  - totalCandy is temporarily transformed to double

    → floating-point division
- In older C, type casting is simpler but confusing

```
candyPerPerson = double(totalCandy)/numberOfPeople;
candyPerPerson = (double) totalCandy/numberOfPeople;
```

Type Cast Example

- int totalCandy = 9, numberOfPeople = 4;
  double candyPerPerson;
  candyPerPerson = static_cast<double>(totalCandy)
  / numberOfPeople;
  - candyPerPerson now is 2.25!
- This would also work:
  candyPerPerson = totalCandy /
  static_cast<double>(numberOfPeople);
- This would not!
  candyPerPerson = static_cast<double>(totalCandy /
  numberOfPeople);

Integer division occurs before type cast

Overview

4.1 Top-Down Design

4.2 Predefined Functions

4.3 Programmer-Defined Functions

4.4 Procedural Abstraction

4.5 Local Variables

4.6 Overloading Function Names

Programmer-Defined Functions

- Two components for a function definition
  - Function declaration (or function prototype)
    - Shows how the function is called
    - Must appear in the code BEFORE the function can be called
    - Syntax:
      Type_returned *Function_Name* (Parameter_List) ;
      //Comment describing what function does
  - Function definition (or function body)
    - Description for real actions in this function
    - Can appear before or after the function is called
    - Syntax:
      Type_returned *Function_Name* (Parameter_List)
      {
      //code to make the function work
      }

Only header is required with an extra semicolon.

Syntax for a Function

Function Declaration

Type_ReturnedFunction_Name (Parameter_List);
Function_Declaration_Comment

Function Definition

Type_Returned Function_Name (Parameter_List)
{
Declaration_1
Declaration_2
…
Declaration_Last

Executable_Statement_1
Executable_Statement_2
…
Executable_Statement_Last
}

Must match to each other

Must include one or more return statements.

Function Declaration

- Function declaration require more information:
  - Tells the return type
  - Tells the name of the function
  - Tells how many arguments are needed
  - Tells the types of the arguments
  - Tells the formal parameter names
    - Formal parameters are like placeholders for the actual arguments used when the function is called
- Example:

```
double totalCost(int numberPar, double pricePar);
// Compute total cost including 5% sales tax on
// numberPar items at cost of pricePar each
```

Function Definition

- Provides the same information as the declaration
- Describes how the function does its task
- Example:

return type

function header

```
double totalCost (int numberPar, double pricePar)
{
    const double TAX_RATE = 0.05; //5% tax
    double subtotal;
    subtotal = pricePar * numberPar;
    return (subtotal + subtotal * TAX_RATE);
}
```

function body

Within a Function Definition

- Variables must be declared before they are used
  - Typically declared before the executable statements begin
- At least one return statement is required
  - Ends the function call and returns the value calculated by the function
  - Each branch of an if-else statement might have its own return statement
- The returning expression can be a constant, a variable holding the calculated value, or a calculation
  - Ex: return subtotal + subtotal * TAX_RATE;
  - Will finish the calculation first and return the result

Example for User-Defined Function

```
//DISPLAY 4.3 A Function Definition
#include <iostream>
using namespace std;
```

Function declaration

```
double totalCost(int numberPar, double pricePar);
//Computes the total cost, including 5% sales tax,
//on numberPar items at a cost of pricePar each.

int main( )
{
    double price, bill;
    int number;

    cout << "Enter the number of items purchased: ";
    cin >> number;
    cout << "Enter the price per item $";
    cin >> price;

    bill = totalCost(number, price);
```

Function call

```
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << number << " items at "
         << "$" << price << " each.\n"
         << "Final bill, including tax, is $" << bill
         << endl;

    return 0;
}
```

Function definition

```
double totalCost(int numberPar, double pricePar)
{
    const double TAX_RATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = pricePar * numberPar;
    return (subtotal + subtotal*TAX_RATE);
}
```

The Function Call

- After the function is well designed (declaration and definition), we can use function call to use it
  - Tells the name of the function and lists the arguments
- Can be used in a statement where the returned value makes sense
  - Ex: double bill = totalCost(number, price);
- The values of the arguments are plugged into the formal parameters by call-by-value mechanism
  - The first argument is used for the first parameter, the second argument for the second parameter, and so forth
  - The value plugged into the formal parameter is used in the function body only
    - The "copy" of input values will not affect the original variables

Function Call Procedure

```
double totalCost(int numberPar, double pricePar)
{
   const double TAX_RATE = 0.05; //5% sales tax
   double subtotal;

   subtotal = pricePar * numberPar;
   return (subtotal + subtotal * TAX_RATE);
}
```

In totalCost function:
numberPar = 2
pricePar = 10.10

20.20 + 20.20*0.05 = 21.21

bill = return value
= 21.21

```
int main()
{
   double price, bill;
   int number;
   cout << "Enter the number of item purchased: ";
   cin >> number;
   cout << "Enter the price per item $";
   cin >> price;

   bill = totalCost(number, price);

   cout.setf(ios::fixed);
   cout.setf(ios::showpoint);
   cout.precision(2);
   cout << number << "item at"
        << "$" << price << "each .\n"
        << "Final bill, including tax, is $" << bill << endl;
   return 0;
}
```

Order of Arguments

- Compiler checks that the types of the arguments are correct and in the correct sequence
  - Function call should match the function declaration
- Compiler cannot check that arguments are in the correct logical order
- Example: char grade(int received_par, int minScore_par);

```
int received = 95,  minScore = 60;

cout <<  grade( minScore,  received);
          ?↓          ↓?
      received_par   minScore_par
```

  - Produces a faulty result because the arguments are not in the correct logical order. The compiler will not catch this!

Example for Ordered Arguments

```
//DISPLAY 4.5 Incorrectly Ordered Arguments
//Determines user's grade. Grades are Pass or Fail.
#include <iostream>
using namespace std;

char grade(int receivedPar, int minScorePar);
//Returns 'P' for passing, if receivedPar is
//minScorePar or higher. Otherwise returns 'F' for failing.

int main( )
{
    int score, needToPass;
    char letterGrade;

    cout << "Enter your score"
         << " and the minimum needed to pass:\n";
    cin >> score >> needToPass;

    letterGrade = grade(needToPass, score);

    cout << "You received a score of " << score << endl
         << "Minimum to pass is " << needToPass << endl;

    if (letterGrade == 'P')
        cout << "You Passed. Congratulations!\n";
    else
        cout << "Sorry. You failed.\n";

    cout << letterGrade
         << " will be entered in your record.\n";

    return 0;
}

char grade(int receivedPar, int minScorePar)
{
    if (receivedPar >= minScorePar)
        return 'P';
    else
        return 'F';
}
```

The three red parts of a function should match to each other !!

Alternate Declarations

- There are two forms for function declarations
  - List formal parameter names
    - Simply copy the header for easy understanding
  - List types of formal parameters, but not names
    - Minimum information required by the compiler
- Examples:

  double totalCost(int numberPar, double pricePar);
  double totalCost(int, double);
- After declaration, add short descriptions about the function in comments
- For function headers, full parameter names are always required!!

Placing Definitions

- A function call must be preceded by either
  - The function's declaration
    or
  - The function's definition
    - If the function's definition precedes the call, a declaration is not needed

```
char grade(int receive, int minScore)
{
   if (receive >= minScore)
      return 'P';
   else
      return 'F';
}

int main( )
{
   ……
   letterGrade = grade(pass, score);
   ……
}
```

- Suggestions to place function declarations:
  - Placing the function declaration prior to the main function (like including library)
  - Placing the function definition after the main function (can be hidden from the main program)
  - This helps to build your own libraries in the future

Boolean Return Values

- A function can return a bool value
  - Can be used where a boolean expression is expected
    - Makes programs easier to read
- if (((rate >=10) && (rate < 20)) || (rate == 0))
  is easier to read as if (appropriate (rate))
  - If function appropriate returns a bool value based on the expression above
- Function appropriate could be defined as

```
bool appropriate(int rate)
{
  return (((rate >=10) && ( rate < 20)) || (rate == 0));
}
```


Default Arguments

- If a function is frequently used with the same argument value for a particular parameter
  → specify that such a parameter has a default value
- When an argument is omitted in a function call, the compiler inserts the default value of that argument
- Default arguments must be the rightmost (trailing) arguments in a function's parameter list
- Default arguments must be specified with the first occurrence of the function name
  - Typically in the function prototype
- Default values can be any expression, including constants, global variables or function calls

Example for Default Arguments (1/2)

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

//function prototype that specifies default arguments
int boxVolume(int length = 1, int width = 1, int height = 1);

int main()
{
   //no arguments--use default values for all dimensions
   cout << "The default box volume is: " << boxVolume();

   //specify length; default width and height
   cout << "\n\nThe volume of a box with length 10,\n"
      << "width 1 and height 1 is: " << boxVolume(10);

   //specify length and width; default height
   cout << "\n\nThe volume of a box with length 10,\n"
      << "width 5 and height 1 is: " << boxVolume(10, 5);
```

Example for Default Arguments (2/2)

```
//specify all arguments
   cout << "\n\nThe volume of a box with length 10,\n"
      << "width 5 and height 2 is: " << boxVolume(10, 5, 2) << endl;
} //end main

int boxVolume(int length, int width, int height)
{
   return length * width * height;
}
```

```
The default box volume is : 1

The volume of a box with length 10,
width 1 and height 1 is : 10

The volume of a box with length 10,
width 5 and height 1 is : 50

The volume of a box with length 10,
width 5 and height 2 is : 100
```

Overview

Procedural Abstraction

- The Black Box Analogy
  - A black box refers to something that we know how to use, but the method of operation is unknown
  - A person using a program needs to know what the program does, not how it does it
  - Do you know how to build a car? You don't have to …

Functions and Black Box Analogy

- A programmer only needs to know what will be produced after arguments are put into the box
- Designing functions as black boxes is an example of information hiding
  - The function can be used without knowing how it is coded
  - The function body can be "hidden from view"
- Designing with the black box in mind allows us
  - Easily change or improve a function without forcing programmers changing what they have done
  - To know how to use a function simply by reading the function declaration and its comment

Example of Changing Black Box

```
double newBalance(double balancePar, double ratePar)
{
    double intersetFraction, interest;

    intersetFraction = ratePar / 100;
    interest = intersetFraction * balancePar;
    return (balancePar + interest);
}
```

same interface but different implementation

```
double newBalance(double balancePar, double ratePar)
{
    double intersetFraction, updateBalance;

    intersetFraction = ratePar / 100;
    updateBalance = balancePar * (1 + intersetFraction);
    return updateBalance;
}
```

Procedural Abstraction and Functions

- Procedural Abstraction is writing and using functions as if they were black boxes
  - Procedure is a general term for a "function like" set of instructions
- Write functions with all a programmer needs
  - Function comment should tell all conditions required of input arguments and returned value
  - Variables used in the function, should be declared in the function body
- Choose meaningful names for formal parameters
  - Formal parameter names are not necessary to match variable names used in the main part of the program
  - Remember that only the value of each argument is passed

Case Study: Buying Pizza

- What size of pizza is the best buy?
  - Which size gives the lowest cost per square inch?
- Input:
  - Diameter of two sizes of pizza
  - Cost of the two sizes of pizza
- Output:
  - Cost per square inch for each size of pizza
  - Which size is the best buy
    - Based on lowest price per square inch
    - If cost per square inch is the same, the smaller size will be the better buy

Buying Pizza: Problem Analysis

- Subtask 1
  - Get the input data for each size of pizza
- Subtask 2
  - Compute price per inch for smaller pizza
- Subtask 3
  - Compute price per inch for larger pizza
- Subtask 4
  - Determine which size is the better buy
- Subtask 5
  - Output the results

identical task?

Buying Pizza: Function Analysis

- Subtask 2 and subtask 3 can be implemented as a single function
  - The calculation for subtask 3 is the same as the calculation for subtask 2 with different arguments
  - Subtask 2 and subtask 3 each return a single value
- Choose an appropriate name for the function
  - double unitprice(int diameter, double price);
    //Returns the price per square inch of a pizza.
    //First parameter is the diameter of the pizze in inches.
    //Second parameter is the price of the pizza.
- Subtasks 2 and 3 are implemented as calls to function *unitprice*
  - Ex: unitPriceSmall = unitprice(diameterSmall, priceSmall);

Buying Pizza: Algorithm Design

- Subtask 1
  - Ask for the input values and store them in variables
    - diameterSmall, priceSmall, diameterLarge, priceLarge
- Subtask 2&3
  - Compute the radius of the pizza
  - Computer the area of the pizza using πr^2
  - Return the value of (price / area)
- Subtask 4
  - Compare cost per square inch of the two pizzas using the less than operator
- Subtask 5
  - Standard output of the results

Buying Pizza: Pseudocode

- Pseudocode
  - Mixture of C++ and English
  - Simplifies algorithm design by ignoring the specific syntax of the programming language
    - If the step is obvious, use C++
    - If the step is difficult to express in C++, use English
  - General guideline: keep the flow control but skip detailed statements
- Ex: pseudocode of function *unitprice*
  - radius = one half of diameter;
    area = Π * radius * radius
    return (price / area)

Buying Pizza: First Try

- double unitprice (int diameter, double price)
  {
  const double PI = 3.14159;
  double radius, area;

  radius = diameter / 2;
  area = PI * radius * radius;
  return (price / area);
  }
  - Oops! Radius should include the fractional part

Buying Pizza: Second Try

- double unitprice (int diameter, double price)
  {
  const double PI = 3.14159;
  double radius, area;

  radius = diameter / static_cast<double>(2);
  area = PI * radius * radius;
  return (price / area);
  }
  - Now radius will include fractional parts
  - radius = diameter / 2.0; // This would also work

Buying Pizza: Code (1/2)

```
//DISPLAY 4.10 Buying Pizza
//Determines which of two pizza sizes is the best buy.
#include <iostream>
using namespace std;

double unitPrice(int diameter, double price);
//Returns the price per square inch of a pizza. The
formal
//parameter named diameter is the diameter of the
pizza in inches.
//The formal parameter named price is the price of the
pizza.

int main( )
{
   int diameterSmall, diameterLarger;

   double priceSmall, unitPriceSmall,priceLarge,
          unitPriceLarge;
```

```
cout << "Welcome to the Pizza Consumers Union.\n";
cout << "Enter diameter of a small pizza (in inches): ";
cin >> diameterSmall;
cout << "Enter the price of a small pizza: $";
cin >> priceSmall;
cout << "Enter diameter of a large pizza (in inches): ";
cin >> diameterLarger;
cout << "Enter the price of a large pizza: $";
cin >> priceLarge;

unitPriceSmall = unitPrice(diameterSmall,
                           priceSmall);
unitPriceLarge = unitPrice(diameterLarger,
                           priceLarge);
```


Buying Pizza: Code (2/2)

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
cout << "Small pizza:\n"
     << "Diameter = " << diameterSmall << " inches\n"
     << "Price = $" << priceSmall
     << " Per square inch = $" << unitPriceSmall <<
          endl
     << "Large pizza:\n"
     << "Diameter = " << diameterLarger << "
          inches\n"
     << "Price = $" << priceLarge
     << " Per square inch = $" << unitPriceLarge <<
          endl;
 if (unitPriceLarge < unitPriceSmall)
     cout << "The large one is the better buy.\n";
 else
     cout << "The small one is the better buy.\n";
     cout << "Buon Appetito!\n";

  return 0;
}
```

```
double unitPrice(int diameter, double price)
{
   const double PI = 3.14159;
   double radius, area;

   radius = diameter/static_cast<double>(2);
   area = PI * radius * radius;
   return (price/area);
}
```

Overview

Local Variables

- Variables declared in a function:
  - Called "local variables" to that function
  - Cannot be used from outside the function
  - Have the function as their scope
- Variables declared in the main part of a program:
  - Are local to the main part of the program
  - Cannot be used from outside the main part
  - Have the main part as their scope
- Local variables cannot be used in another functions. Only their values can be passed
  - Passed-by-value mechanism

Global Variables

- Global variables
  - Available to more than one function as well as the main part of the program
  - Declared outside any function body (including main)
- Can be used when more than one function must use a common variable
  - Generally make programs more difficult to understand and maintain
- Ex:
```
const double PI = 3.14159;
double volume(double);
int main()
{...}
```
  - PI is available to main function and function volume

Code Example for Global Constant

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

const double PI = 3.14159;
double area(double radius);
double volume(double radius);

int main( )
{
   double radiusOfBoth, areaOfCircle, volumeOfSphere;

   cout << "Enter a radius to use for both a circle\n"
        << "and a sphere (in inches): ";
   cin >> radiusOfBoth;

   areaOfCircle = area(radiusOfBoth);
   volumeOfSphere = volume(radiusOfBoth);
   cout << "Radius = " << radiusOfBoth << " inches\n"
        << "Area of circle = " << areaOfCircle
        << " square inches\n"
        << "Volume of sphere = " << volumeOfSphere
        << " cubic inches\n";

   return 0;
}
```

```
double area(double radius)
{
    return (PI * pow(radius, 2));
}

double volume(double radius)
{
    return ((4.0/3.0) * PI * pow(radius, 3));
}
```

Sample Dialogue

```
Enter a radius to use for both a circle
and a sphere (in inches): 2
Radius = 2 inches
Area of circle = 12.5664 square inches
Volume of sphere = 33.5103 cubic inches
```

Formal Parameters are Local

- Formal parameters are actually local variables to the function definition
  - Just as if they were declared in the function header
  - Do NOT re-declare the formal parameters in the function body → double declaration
- Call-by-value mechanism gives the initial values
  - When a function is called, the formal parameters are initialized to the given values from the function call
  - The formal parameters can be altered later in the function, but has no impact to the original variables

Formal Parameters Used as Local

Sample Dialogue

```
Welcome to the offices of
Dewey, Cheatham, and Howe.
The law office with a heart.
Enter the hours and minutes of your consultation:
2 45
For 2 hours and 45 minutes, your bill is $1650.00
```

```
//DISPLAY 4.13: Law office billing program.
#include <iostream>
using namespace std;

const double RATE = 150.00;
double fee(int hoursWorked, int minutesWorked);

int main( )
{
   ……
   bill = fee(hours, minutes);
   ……
}
```

```
                                  2            45
double fee(int hoursWorked, int minutesWorked)
{
   int quarterHours;
                                                  2*60+45=165
   minutesWorked = hoursWorked*60 +
                   minutesWorked;                 165/15=11
   quarterHours = minutesWorked/15;
   return (quarterHours*RATE);
}                                                 11*150=1650
```

Block Scope

- Local and global variables conform to the rules of Block Scope
  - The code block (generally defined by the { }) determines the scope of the identifiers declared in it
  - Blocks can be nested
- Different functions may use different namespaces
  - Placing *using namespace std* inside the starting brace of a function
    - Allows the use of different namespaces in different functions
    - Makes the "using" directive local to the function

Example for Block Scope

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

4 const double GLOBAL_CONST = 1.0;
5 int function1(int param);

7 int main()
8 {
9    int x;
10   double d = GLOBAL_CONST;

12   for (int i = 0; i < 10; i++)
13   {
14       x = function1(i);
15   }
16   return 0;
17 }

19 int function1(int param)
20 {
21    double y = GLOBAL_CONST;
22    ...
23    return 0;
24 }
```

Block scope: Variable i has scope from lines 12-15

Local scope to **main**: Variable x has scope from lines 9-17 and variable d has scope from lines 10-17

Local scope to **function1**: Variable **param** has scope from lines 19-24 and variable y has scope from lines 21-24

Global scope: The constant **GLOBAL_CONST** has scope from lines 4-24 and the function **function1** has scope from lines 5-24

Example of Using Namespaces

```
//DISPLAY 4.15 Using Namespaces
#include <iostream>
#include <cmath>

const double PI = 3.14159;

double area(double radius)
{
    using namespace std;

   return (PI * pow(radius, 2));
}
//Returns the area of a circle with the specified
radius.

double volume(double radius)
{
    using namespace std;

   return ((4.0/3.0) * PI * pow(radius,3));
}
//Returns the volume of a sphere with the specified
radius.
```

```
int main( )
{
    using namespace std;

   double radiusOfBoth, areaOfCircle, volumeOfSphere;

   cout << "Enter a radius to use for both a circle\n"
        << "and a sphere (in inches): ";
   cin >> radiusOfBoth;

   areaOfCircle = area(radiusOfBoth);
   volumeOfSphere = volume(radiusOfBoth);

   cout << "Radius = " << radiusOfBoth << " inches\n"
        << "Area of circle = " << areaOfCircle
        << " square inches\n"
        << "Volume of sphere = " << volumeOfSphere
        << " cubic inches\n";

   return 0;
}
```


Example: Factorial

- n! represents the factorial function
- n! = 1 x 2 x 3 x … x n
- The C++ version of the factorial function found in Display 3.14
  - Requires one argument of type int, n
  - Returns a value of type int
  - Uses a local variable to store the current product
  - Decrements n each time for another multiplication
    n * n-1 * n-2 * … * 1

Code for Function Factorial

```
//DISPLAY 4.16 Factorial Function
//Function Declaration
int factorial(int n);
//Returns factorial of n.
//The argument n should be nonnegative.

//Function Definition
int factorial(int n)
{
    int product = 1;
    while (n > 0)
    {
        product = n * product;
        n--;
    }

    return product;
}
```

n becomes a variable that can be assigned to different value each time

Overview

Overloading Function Names

- C++ allows more than one definition for the same function name
  - Very convenient when the "same" function is needed for different numbers or types of arguments
- If there are more than one definition using the same function name, how to choose correct one?
  - Clear rules are required to make decisions automatically
- Requirements for overloaded functions
  - Must have different numbers of formal parameters AND / OR
  - Must have at least one different type of parameter
  - Must return a value of the same type

Overloading Examples

- double ave(double n1, double n2)
  {
  return ((n1 + n2) / 2);
  }
- double ave(double n1, double n2, double n3)
  {
  return ((n1 + n2 + n3) / 3);
  }
- Compiler checks the number and types of arguments in the function call to decide which function to use

  cout << ave(10, 20, 30);

  uses the second definition because of the argument numbers

Revisit the Pizza Buying program

- Rectangular pizzas are now offered!
- Change the input and add a function to compute the unit price of a rectangular pizza
- The new function could be named unitprice_rectangular
- Or, the new function could be a new (overloaded) version of the *unitprice* function that is already used
  - Example:

```
double unitprice(int length, int width, double price)
  {
      double area = length * width;
      return (price / area);
  }
```

Example of Overloading (1/2)

```
//DISPLAY 4.18 Overloading a Function Name
//Determines whether a round pizza or a rectangular
pizza is the best buy.
#include <iostream>

double unitPrice(int diameter, double price);
//Returns the price per square inch of a round pizza.
//The formal parameter named diameter is the diameter
of the pizza
//in inches. The formal parameter named price is the
price of the pizza.

double unitPrice(int length, int width, double
price);
//Returns the price per square inch of a rectangular
pizza
//with dimensions length by width inches.
//The formal parameter price is the price of the pizza.
```

```
int main( )
{
    using namespace std;
    int diameter, length, width;
    double priceRound, unitPriceRound,
          priceRectangular, unitPriceRectangular;

    cout << "Welcome to the Pizza Consumers
              Union.\n";
    cout << "Enter the diameter in inches"
         << " of a round pizza: ";

    cin >> diameter;
    cout << "Enter the price of a round pizza: $";

    cin >> priceRound;
    cout << "Enter length and width in inches\n"
         << "of a rectangular pizza: ";

    cin >> length >> width;
    cout << "Enter the price of a rectangular pizza: $";

    cin >> priceRectangular;
```

Example of Overloading (2/2)

```
unitPriceRectangular =
        unitPrice(length, width, priceRectangular);
unitPriceRound = unitPrice(diameter, priceRound);

cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
cout << endl
     << "Round pizza: Diameter = "
     << diameter << " inches\n"
     << "Price = $" << priceRound
     << " Per square inch = $" << unitPriceRound
     << endl
     << "Rectangular pizza: Length = "
     << length << " inches\n"
     << "Rectangular pizza: Width = "
     << width << " inches\n"
     << "Price = $" << priceRectangular
     << " Per square inch = $ "
     << unitPriceRectangular
     << endl;
```

```
if (unitPriceRound < unitPriceRectangular)
     cout << "The round one is the better buy.\n";
else
     cout << "The rectangular one is the better buy.\n";
     cout << "Buon Appetito!\n";
return 0;
}

double unitPrice(int diameter, double price)
{
   const double PI = 3.14159;
   double radius, area;

   radius = diameter/static_cast<double>(2);
   area = PI * radius * radius;
   return (price/area);
}

double unitPrice(int length, int width, double price)
{
   double area = length * width;
   return (price/area);
}
```

Type Conversion Problem

- Given the definition

```
double mpg(double miles, double gallons)
{
     return (miles / gallons);
}
```

The arguments are converted to type double (45.0 and 2.0)

What will happen if mpg is called in this way?

```
cout << mpg(45, 2) << " miles per gallon";
```

- Given another mpg definition in the same program

```
int mpg(int goals, int misses)  // the Measure of Perfect Goals
{
     return (goals – misses);
}
```

Compiler chooses this function because the parameter types match

What happens if mpg is called this way now?

```
cout << mpg(45, 2) << " miles per gallon";
```

- Do not use the same function name for unrelated functions!!