

Chapter 4

Procedural Abstraction and Functions That Return a
Value

Overview

- Predefined Functions
- Programmer-Defined Functions
 - Local Variables
 - Overloading Function Names
- Top-Down Design and Procedural Abstraction

C++ Functions

- Function: a **packaged** segment of code (i.e., **has a name**) that implement a well-defined functionality, e.g.,
 - To calculate square root of a double value, named *sqrt*
 - To read from keyboard the number of items to order, named *GetItemNum*,
 - To calculate area of a circle with a given radius
- **Blackbox analogy**: a function takes some input and produces some output
 - Input: the value to calculate square root of, the item's name, radius
 - Output: the result of calculation, or user's input (as in *GetItemNum*)
 - Input provided as parameters, output as return value ...

Program structure

- Function declaration, function definition, function call
 - Syntax, formal parameters, actual arguments
 - Function must be declared before being called

```
int GetItemNumber (string itemName);
```

function declaration

```
int main( )
```

```
{
```

actual arguments

```
    GetItemNumber ("14-inch pizza");
```

function call

```
}
```

```
int GetItemNumber (string itemName)
```

formal parameters

```
{
```

```
    int value;
```

```
    cout << "Enter the number of item " << itemName << ":";
```

```
    cin >> value;
```

```
    return value;
```

function definition

function call

- You can call a function from main, from any function (including the function itself, i.e., recursive function)
 - `value = sqrt (9.0)+12.0;`
 - `area = AreaCircle (12.00);`
 - `cout << "Area of a circle with diameter "<< diameter
 << " is " <<AreaCircle (diameter/2.0);`
 - Note: if you declare and define a function, but do not call the function from `main()`, the function will not be executed

function call

- Parameter passing (pass-by-value): the values of arguments are **assigned** to parameters **based on order**
 - Normally, arguments can be expressions, variables, or constants
 - Type of arguments should match with corresponding parameters (recall they will be assigned to the parameters)
- Return from function:
 - When reach return statement, or end of function body
 - Go back to caller function, to the statement where function is called

Predefined Functions

- C++ comes with libraries of predefined functions
- Example: `sqrt` function that returns, or computes, square root of a number
 - `the_root = sqrt(9.0);`
 - the number, 9, is called the **argument**
 - `the_root` will contain 3.0
- `sqrt(9.0)` is a **function call**
 - It invokes, or sets in action, the `sqrt` function
 - argument (here it is 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);`

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;
}
```

Sample Dialogue

Enter the amount budgeted for your dog house **\$25.00**
For a price of \$25.00
I can build you a luxurious square dog house
that is 1.54 feet on each side.

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);`

Function Libraries

- Predefined functions are found in libraries
- The library must be “included” in a program to make the functions available
- An include directive tells the compiler which library header file to include.
- To include the math library containing `sqrt()`:

`#include <cmath>`

- Newer standard libraries, such as `cmath`, also require directive `using namespace std;`

Other Predefined Functions

- `abs(x)` `--- int value = abs(-8);`
 - Returns absolute value of argument `x`
 - Return value is of type `int`
 - Argument is of type `x`
 - Found in the library `cstdlib`
- `fabs(x)` `--- double value = fabs(-8.0);`
 - Returns the absolute value of argument `x`
 - Return value is of type `double`
 - Argument is of type `double`
 - Found in the library `cmath`

Some Predefined Functions

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

Random Number Generation

- Really pseudo-random numbers
- 1. Seed the random number generator only once

```
#include <stdlib>
```

```
#include <ctime>
```

```
srand(time(0));
```

- 2. `rand()` function returns a random integer that is greater than or equal to 0 and less than `RAND_MAX`

```
rand();
```

Random Numbers

- Use % and + to scale to the number range you want
- For example to get a random number from 1-6 to simulate rolling a six-sided die:

```
int die = (rand() % 6) + 1;
```

- Can you simulate rolling two dice?
- Generating a random number x where $10 < x < 21$?

Type Casting

- Recall the problem with integer division:
int total_candy = 9, number_of_people = 4;
double candy_per_person;
candy_per_person = total_candy / number_of_people;
 - *candy_per_person = 2, not 2.25!*
- A Type Cast produces a value of one type from another type
 - *static_cast<double>(total_candy)* produces a double representing integer value of *total_candy*

Type Cast Example

- [illegible]

Integer division occurs before type cast

Old Style Type Cast

- C++ is an evolving language
- This older method of type casting may be discontinued in future versions of C++

candy_per_person = double(total_candy)/number_of_people;

Exercises

- Determine the value of d?

`double d = 11 / 2;`

- Determine the value of

`pow(2,3) fabs(-3.5) sqrt(pow(3,2))`
`7 / abs(-2) ceil(5.8) floor(5.8)`

- Convert the following to C++

$$\sqrt{x + y}$$

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Overview

- Predefined Functions
- **Programmer-Defined** Functions
 - Local Variables
 - Overloading Function Names
- Top-Down Design and Procedural Abstraction

Programmer-Defined Functions

- Two components of 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**
 - Describes *how the function does its task*
 - Can appear before or after the function is called
 - Syntax:

```
Type_returned Function_Name(Parameter_List)  
{  
    // code to make the function work  
}
```

Function Declaration

- Example:

```
double total_cost(int number_par, double price_par);  
// Compute total cost including 5% sales tax on  
// number_par items at cost of price_par each
```

- Tell compiler about *total_cost* function:
 - the return type
 - the name of the function
 - how many arguments are needed
 - the types of the arguments
 - the formal parameter names
 - Formal parameters are like placeholders for the actual arguments used when the function is called
 - Formal parameter names can be any valid identifier

Alternate Declarations

- Two forms of function declaration:

```
double total_cost(int number_par, double price_par);
```

```
// List formal parameter names
```

```
// Compute total cost including 5% sales tax on
```

```
// number_par items at cost of price_par each
```

```
double total_cost(int, double);
```

```
// List types of formal parameters, but not names
```

```
// The first parameter is number of items, and the second is the unit price
```

```
// Compute total cost including 5% sales tax on the specified number of
```

```
// items at given unit price
```

- First aids description of the function in comments
 - Function headers must always list formal parameter names!

Placing Definitions

- A function call must be preceded by either
 - The function's declaration, [see example code](#)
or
 - The function's definition, [see example code](#)
 - If the function's definition precedes the call, a declaration is not needed
- Placing the function declaration prior to main function and the function definition after the main function leads naturally to building your own libraries in the future.

Function Definition

- Function header: Provides the same information as the declaration
- **Function body:** Describes how the function does its task
- Example:

```
double total_cost(int number_par, double price_par)
{
    const double TAX_RATE = 0.05; // 5% tax
    double subtotal;
    subtotal = price_par * number_par;
    return (subtotal + subtotal * TAX_RATE);
}
```

function header

function body

- Within a function definition
 - Variables must be declared before they are used
 - Variables are typically declared before the executable statements begin
 - At least one return statement must end the function
 - Each branch of an if-else statement might have its own return statement

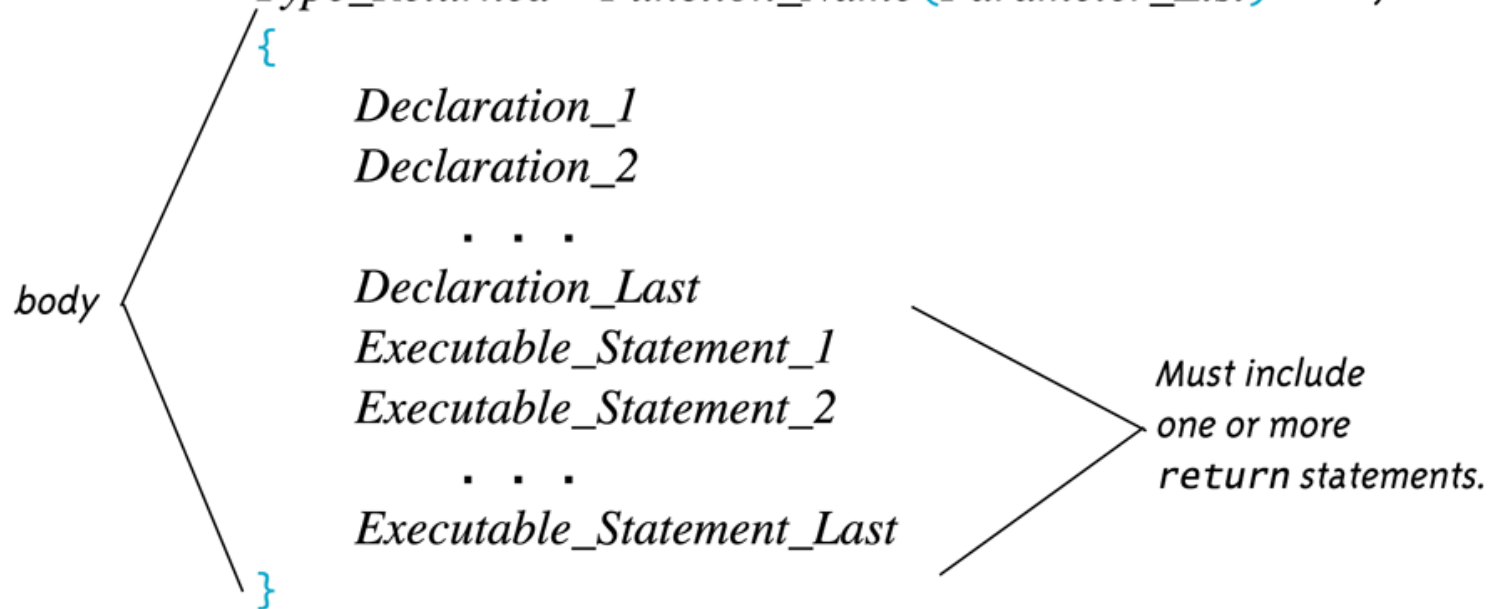
Syntax for a Function That Returns a Value

Function Declaration

Type_Returned *Function_Name* (*Parameter_List*);
Function_Declaration_Comment

Function Definition

Type_Returned *Function_Name* (*Parameter_List*) ← function header



Return Statement

- Ends the function call
- Returns the value calculated by the function
- Syntax:

return expression;

- expression performs the calculation or a variable containing the calculated value
- Example:

*area=PI*radius*radius;*

return area;

or

*return PI*radius*radius;*

Overview

- Predefined Functions
- Programmer-Defined Functions
 - Local Variables
 - Overloading Function Names
- Top-Down Design and Procedural Abstraction

Local Variables

- Variables declared in a function:
 - Are local to that function, they cannot be used from outside function
 - Have the function as their scope
- Variables declared in the **main function** of a program:
 - Are local to the main function, they cannot be used from outside main function
 - Have the main part as their scope

Formal Parameters are Local Variables

- **Formal Parameters** are actually variables that are local to the function definition
 - They are used just as if they were declared in function body
 - Do NOT re-declare formal parameters in function body, they are declared in the function declaration
- call-by-value mechanism
 - When a function is called, formal parameters are initialized to values of the arguments in the function call

Local Variables (part 1 of 2)

```
//Computes the average yield on an experimental pea growing patch.
#include <iostream>
using namespace std;

double est_total(int min_peas, int max_peas, int pod_count);
//Returns an estimate of the total number of peas harvested.
//The formal parameter pod_count is the number of pods.
//The formal parameters min_peas and max_peas are the minimum
//and maximum number of peas in a pod.

int main()
{
    int max_count, min_count, pod_count;
    double average_pea, yield;

    cout << "Enter minimum and maximum number of peas in a pod: ";
    cin >> min_count >> max_count;
    cout << "Enter the number of pods: ";
    cin >> pod_count;
    cout << "Enter the weight of an average pea (in ounces): ";
    cin >> average_pea;

    yield =
        est_total(min_count, max_count, pod_count) * average_pea;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(3);
    cout << "Min number of peas per pod = " << min_count << endl
        << "Max number of peas per pod = " << max_count << endl
        << "Pod count = " << pod_count << endl
        << "Average pea weight = "
        << average_pea << " ounces" << endl
        << "Estimated average yield = " << yield << " ounces"
        << endl;

    return 0;
}
```

*This variable named
average_pea is local to the
main part of the program.*

Local Variables (part 2 of 2)

```
double est_total(int min_peas, int max_peas, int pod_count)
{
    double average_pea;

    average_pea = (max_peas + min_peas)/2.0;
    return (pod_count * average_pea);
}
```

*This variable named
average_pea is local to
the function est_total.*

Sample Dialogue

Enter minimum and maximum number of peas in a pod: 4 6
Enter the number of pods: 10
Enter the weight of an average pea (in ounces): 0.5
Min number of peas per pod = 4
Max number of peas per pod = 6
Pod count = 10
Average pea weight = 0.500 ounces
Estimated average yield = 25.000 ounces

Global Variables

- Global Variable -- used when more than one function must use a common variable
 - Available to more than one function as well as main part of the program
 - Declared outside any function body
 - Declared outside the main function body
 - Declared before any function that uses it
- Generally make programs more difficult to understand and maintain

Global Constants

- Global Named Constant
 - A global variable that is constant
- Example:

```
const double PI = 3.14159;  
double volume(double);
```

```
int main()  
{  
    ...  
}  
...
```

- PI is available to the main function and to function *volume*

A Global Named Constant (part 1 of 2)

```
//Computes the area of a circle and the volume of a sphere.
//Uses the same radius for both calculations.
#include <iostream>
#include <cmath>
using namespace std;

const double PI = 3.14159;

double area(double radius);
//Returns the area of a circle with the specified radius.

double volume(double radius);
//Returns the volume of a sphere with the specified radius.

int main()
{
    double radius_of_both, area_of_circle, volume_of_sphere;

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

    area_of_circle = area(radius_of_both);
    volume_of_sphere = volume(radius_of_both);

    cout << "Radius = " << radius_of_both << " inches\n"
         << "Area of circle = " << area_of_circle
         << " square inches\n"
         << "Volume of sphere = " << volume_of_sphere
         << " cubic inches\n";

    return 0;
}
```

demo_pi.cpp

A Global Named Constant (part 2 of 2)

```
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 Parameter Used as a Local Variable (part 1 of 2)

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

const double RATE = 150.00; //Dollars per quarter hour.

double fee(int hours_worked, int minutes_worked);
//Returns the charges for hours_worked hours and
//minutes_worked minutes of legal services.

int main()
{
    int hours, minutes;
    double bill;

    cout << "Welcome to the offices of\n"
         << "Dewey, Cheatham, and Howe.\n"
         << "The law office with a heart.\n"
         << "Enter the hours and minutes"
         << " of your consultation:\n";
    cin >> hours >> minutes;

    bill = fee(hours, minutes);


    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "For " << hours << " hours and " << minutes
         << " minutes, your bill is $" << bill << endl;

    return 0;
}

double fee(int hours_worked, int minutes_worked)
{
    int quarter_hours;

    minutes_worked = hours_worked*60 + minutes_worked;
    quarter_hours = minutes_worked/15;
    return (quarter_hours*RATE);
}
```

*The value of minutes
is not changed by the
call to fee.*



Formal Parameter Used as a Local Variable (part 2 of 2)

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

*minutes_worked is
a local variable
initialized to the
value of minutes.*

Block Scope

- **A block, or block statement, or code block:** C++ code/statements enclosed in braces
 - **Global block:** the outermost block that encompasses all code in a program
 - Blocks can be nested: function block within global block, a for loop block within a function block, ...
- **Block scope rule:** identifier (variable) declared within a block is local to the block, i.e., only accessible from point of declaration to end of block
 - Ex: local and global variables

Block Scope Revisited

```
1  #include <iostream>
2  using namespace std;
3
4  const double GLOBAL_CONST = 1.0;
5
6  int function1 (int param);
7
8  int main()
9  {
10     int x;
11     double d = GLOBAL_CONST;
12
13     for (int i = 0; i < 10; i++)
14     {
15         x = function1(i);
16     }
17     return 0;
18 }
19
20 int function1 (int param)
21 {
22     double y = GLOBAL_CONST;
23     ...
24     return 0;
25 }
```

Local and Global scope are examples of Block scope.
A variable can be directly accessed only within its scope.

Block scope:
Variable **i** has
scope from
lines 13-16

Local scope to
main: Variable
x has scope
from lines
10-18 and
variable **d** has
scope from
lines 11-18

Global scope:
The constant
GLOBAL_CONST
has scope from
lines 4-25 and
the function
function1
has scope from
lines 6-25

Local scope to **function1**:
Variable **param**
has scope from lines 20-25
and variable **y** has scope
from lines 22-25

Exercises

- Which of the following statements about variables is true?
 1. The same variable name can be used in two different functions.
 2. The same variable name cannot be used for two different variables in a single function.
 3. You should use global variables whenever possible.
 4. A variable is visible from the point at which it is defined until the end of the program.

Exercises

- Which of the following is correct about a global variable?
 1. It is declared before all the functions in a program.
 2. It is visible to all the functions declared after it.
 3. It is declared in the main function.
 4. It is declared within the scope of a function.

Brush up your emacs skill

How to cut and paste?

Emacs keystroke notations

- **Ctrl key:** **C- prefix** to denote holding down Ctrl while pressing another key.
 - holding down Ctrl and pressing `x` is denoted as `C-x`
- **Meta key:** **M- prefix**, pressing Esc once and releasing it
 - `M-x`, press Esc and release it, then press `x`
- **M-C- prefix:** press and release Esc, then hold down Ctrl while pressing the final key in command.
- **Examples**
 - `C-x C-c`: Press and hold Ctrl, then press `x` , followed by `c` .
 - `M-x shell`: Press and release Esc, then press `x` . Then type shell and press Enter.
 - `M-C-p`: Press and release Esc, then press and hold Ctrl and press `p` .

undo

- *Undo*: reverses recent changes (not yet saved to file)
 - You can undo all the changes one at a time
- Keystrokes for undo
 - C-/: press ctrl and / together
 - C-x u: press ctrl and x together, and then u
 - C-_: press ctrl and space together

Other emacs commands

Useful commands

M-f	next word	M-b	previous word
C-a	beginning of line	C-e	end of line
C-k	delete line (starting from cursor)	C-d	delete next character after the cursor
C-y	restore (yank) line	C-x i	insert (file)
M-<	top of file	M->	bottom of file
C-v	next screen	M-v	previous screen
C-s	search	M-x	and then type "goto-line"
C-x C-s	save file	C-x C-c	quit emacs
C-z	suspend emacs (fg to restore)	C-x C-w	save file to different name

How to cut/copy/ paste ?

- First **select** part of texts that you want to cut or copy
 - First set a mark at one end, then move cursor to other end.
 - To set mark, press C-SPC or C-@
 - Region is all of the text from mark to cursor.
- Then **cut or copy**
 - To cut selected text, press **C-w** (Ctrl-w)
 - To copy selected text, press **M-w** (Press Esc, release it, and then press w)
- To **paste** text that has just been cut/copied
 - press **C-y** (Ctrl-y)

Overview

- Predefined Functions
- Programmer-Defined Functions
 - Local Variables
 - Function call: how to trace?
 - Overloading Function Names
- Top-Down Design and Procedural Abstraction

Recall handtracing

- Purpose: identify logic errors, understand others' code, verify your code/design
- What you needs:
 - The program
 - Paper and pencil
- Keys:
 - **Arrows**: pointing to current statement
 - **A variable table**: keeping track of variable values
 - **Input region**: what have been entered by user in keyboard, what have been read by the program?
 - **Output region**: keeping track of information displayed in terminal (standard output)

Tracing Function Call

- Create a new variable table for function's local variables
 - Include **formal parameters**, those listed inside (...)
- **Parameter Passing**: assign values of arguments to formal parameters **based on order** (not by names...)
 - first argument assigned to first formal parameter
 - second argument for second formal parameter, and so forth
 - So called “**pass-by-value**” parameters
- Start to execute function body from first statement, until a **return** statement, or **}** (end of function)
- Value **returned to the caller, and resume execution** in caller function (at the point where function is called)

DISPLAY 4.4 Details of a Function Call

```
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 = total_cost (number, price);

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

1. Before the function is called, values of the variables **number** and **price** are set to **2** and **10.10**, by cin statements (as you can see the Sample Dialogue in Display 4.3)

2. The function call executes and the value of **number** (which is 2) plugged in for **number_par** and value of **price** (which is 10.10) plugged in for **price_par**.

```
double total_cost (int number_par, double price_par)
{
    const double TAX_RATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = price_par * number_par;
    21.21 return (subtotal + subtotal*TAX_RATE);
}
```

3. The body of the function executes with **number_par** set to 2 and **price_par** set to 10.10, producing the value 20.20 in subtotal.

4. When the return statement is executed, the value of the expression after return is evaluated and returned by the function. In this case, (**subtotal + subtotal * TAX_RATE**) is (**20.20 + 20.20*0.05**) or 21.21.

5. The value 21.21 is returned to where the function was invoked. The result is that **total_cost(number, price)** is replaced by the return value of 21.21. The value of **bill** (on the left-hand side of the equal sign) is set equal to 21.21 when the statement **bill = total_cost (number, price);** finally ends.

Example: Factorial

- $n!$ Represents the factorial function

$$n! = 1 \times 2 \times 3 \times \dots \times n$$

- C++ version of the factorial function
 - Requires one argument of type `int`, `n`
 - Returns a value of type `int`
- Algorithm
 - Uses a local variable to store current product
 - Decrements `n` each time it does another multiplication


$$n * n-1 * n-2 * \dots * 1$$

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--;  formal parameter n  
    }  
  
    return product;  
}
```

```
int main ()
```

```
{
```



```
int k;
```

```
cout << "Enter a natural number:";
```

```
cin >> k;
```

```
int result = factorial (k);
```

```
cout << "factorial of " << k << "="
```

```
<< result << endl;
```

```
}
```

```
int factorial(int k)
```

```
{
```

```
int product = 1;
```

```
while (k > 0)
```

```
{
```

```
product = k * product;
```

```
k--;
```

```
}
```

```
return product;
```

```
}
```

Factorial.cpp

main() local variables

k			
result			

factorial(k) local variables

k			
product			

Input/Output:

Enter a natural number: 4

Trace with help of computer

- Add *cout* statements to trace the execution of program
 - Before and after each chunk of code
 - before and after loop
 - Before and after function calls
 - Inside function call: entry point and exit point
 - To make sure parameters have been passed correctly
 - Anywhere you suspect something has gone wrong:
 - Binary search to identify where errors occur

Example: `fact_trace.cpp`

- You can also add some “break points”:
 - Example: `fact_trace2.cpp`
- Tools: gdb or IDE to trace your program

Order of Arguments

- Compiler checks that types of arguments are correct and in the correct sequence.
- Compiler cannot check that arguments are in the correct logical order
- Example: Given the function declaration:
`char grade(int received_par, int min_score_par);`

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

```
cout << grade( min_score, received);
```

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

Incorrectly Ordered Arguments (part 1 of 2)

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

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

int main()
{
    int score, need_to_pass;
    char letter_grade;

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

    letter_grade = grade(need_to_pass, score);

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

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

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

    return 0;
}

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

grade.cpp

Although formal parameter names only matters in terms of style (like indentation), good names prevent such mistakes.

Imagine this and no comment:

char grade (int a, int b);

Automatic Type Conversion

- Given the definition

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

what will happen if mpg is called in this way?

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

- values of arguments will automatically be converted to type double (45.0 and 2.0)

bool Return Values

- A function can return a bool value
 - Such a function can be used where a boolean expression is expected

```
if (((rate >= 10) && (rate < 20)) || (rate == 0))  
    ...
```

```
if (appropriate (rate))
```

```
    ...
```

```
bool appropriate(int rate)
```

```
{
```

```
    return (((rate >= 10) && (rate < 20)) || (rate == 0));
```

```
}
```

Example

- Write a function for testing leap year, it takes the year as parameter, and return true if the year is leap year; and return false otherwise.
 - Start with declaration, comment (a kind of contract, requirement analysis: **what does this function do?**)
 - Then move on to definition (i.e., actual implementation: **how does the function achieve it?**)

Example

- Write a function that returns the number of days in a month.
 - Function declaration:
 - What parameter(s) (i.e., input) the function needs?
 - What output the function generates (i.e., return value)?
 - Function definition:

Overview

- Predefined Functions
- Programmer-Defined Functions
 - Local Variables
 - Function call: how to trace?
 - **Overloading Function Names**
- Top-Down Design and Procedural Abstraction

Overloading Function Names

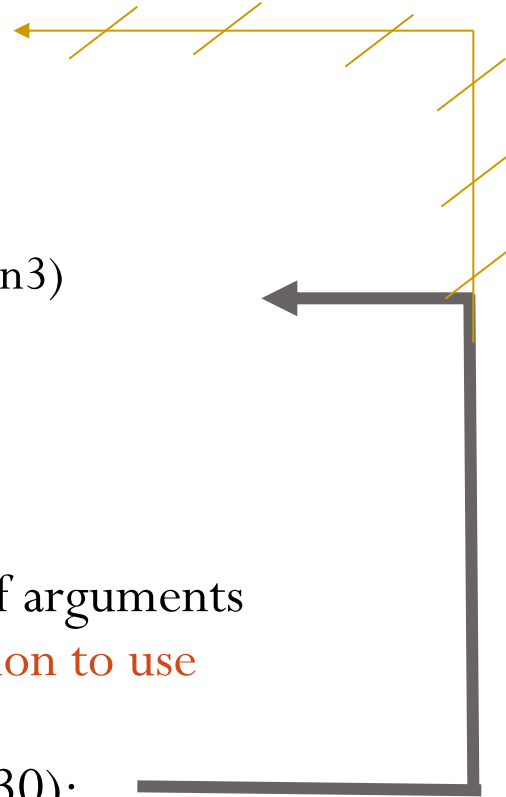
- C++ allows more than one definition for same function name
 - Very convenient for situations in which “same” function is needed for different numbers or types of arguments
 - E.g., average of integers, average of doubles
 - E.g., area of circle, area of rectangle, ...
- **Overloading** a function name: more than one declaration and definition using same function name

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 number and types of arguments
 in function call to **decide which function to use**

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

uses the second definition





Overloading Details

- Overloaded functions
 - **Must return a value of the same type**
 - Must have different numbers of formal parameters, or at least one parameter is of different type
- Example: `overload_test.cpp`

Overloading a Function Name

```
//Illustrates overloading the function name ave.  
#include <iostream>  
  
double ave(double n1, double n2);  
//Returns the average of the two numbers n1 and n2.
```

```
double ave(double n1, double n2, double n3);  
//Returns the average of the three numbers n1, n2, and n3.
```

```
int main()  
{  
    using namespace std;  
    cout << "The average of 2.0, 2.5, and 3.0 is "  
        << ave(2.0, 2.5, 3.0) << endl;  
  
    cout << "The average of 4.5 and 5.5 is "  
        << ave(4.5, 5.5) << endl;  
  
    return 0;  
}  
  
double ave(double n1, double n2)   
{  
    return ((n1 + n2)/2.0);  
}  
  
double ave(double n1, double n2, double n3)   
{  
    return ((n1 + n2 + n3)/3.0);  
}
```

overload_ex.cpp

Output

```
The average of 2.0, 2.5, and 3.0 is 2.50000  
The average of 4.5 and 5.5 is 5.00000
```



```
double mpg(double miles, double gallons) //return miles per gallon
```

```
{
```

```
    return (miles / gallons);
```

```
}
```

```
int mpg(int goals, int misses)
```

```
// returns the Measure of Perfect Goals
```

```
{
```

```
    return (goals — misses);
```

```
}
```

Function overloading mpg.cpp

what happens if mpg is called as follows?

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

- Compiler chooses function that matches parameter types so Measure of Perfect Goals will be calculated

Do not use the same function name for unrelated functions

Overview

- Predefined Functions
- Programmer-Defined Functions
 - Local Variables
 - Function call: how to trace?
 - Overloading Function Names
- Why functions?
 - Procedural Abstraction
 - How does it help with problem solving, i.e., Top-Down Design

Without functions

- If there is no function
 - We need to use C++ statements to write codes to calculate square root, exponentials, logarithm, write to a disk file, write to terminal
 - Everything will be located in a super long main()
 - We have to copy code around, as we might need to calculate square root in multiple locations in the code
- Hard to re-use code:
 - if a friend of yours implemented calculating square root, you have to copy/embed his/her code in your main, making sure there is no name collision
- Code: hard to read, understand, maintain !

Function to rescue

- We “**package**” codes that implement a well-defined functionalities into a “function”
 - For example, code to calculate square root
 - To read an order of an item
- We then call the function from multiple places
 - `GetItemNumber (“12-inch pizza”);`
 - `GetItemNumber (“14-inch pizza”);`
- Share it with others, therefore we have `iostream`, `cmath` library which are a collection of function definitions
`#include <cmath> //add math function declarations into your program, so you can call them...`
- Change function’s implementation without affect callers

Procedural Abstraction

- **Black Box:** refers to something that we know how to use (inputs, buttons, menus and output), but the method of operation is unknown
 - A person using a program does not need to know how it is coded, but he know what the function does
 - E.g., you know brake is for slowing down, but do not know how it works ...
- Functions and Black Box Analogy
 - A programmer who uses a function needs to know what the function does, not how it does it
 - Only function header, and comment (Header: specify input and output of the function)
 - E.g., you do not know how `sqrt()` is implemented ...

Procedural Abstraction and C++

- **Procedural Abstraction** is writing and using functions as if they were black boxes
 - Procedure is a general term meaning a “function like” set of instructions
 - **Abstraction:** when you use a function as a black box, you abstract away details of code in the function body

Guideline for function design

- Each function does one well-defined thing
 - Do not do multiple things in one function, e.g., read item number and calculate the cost ...
 - And possibly needed for future
 - One-liner, 2-liners are not well suited for function
- In general, return the result, instead of cout the result
 - Let the caller function decides what to do with
- **Information hiding**: function can be used without knowing how it is coded (function body can be “hidden from view”), **declaration and comment** is all a programmer needs to know
 - comment should tell all conditions required of parameters to function, and choose meaningful names for formal parameters,
 - Comment should describe the returned value

Overview

- Predefined Functions
- Programmer-Defined Functions
 - Local Variables
 - Function call: how to trace?
 - Overloading Function Names
- Why functions?
 - Procedural Abstraction
 - How does it help with problem solving, i.e., Top-Down Design

Top Down Design

- To write a program
 - Develop **algorithm** that program will use
 - Translate algorithm into programming language
- Top Down Design (also called **stepwise refinement**)
 - Break the algorithm into subtasks
 - Break each subtask into smaller subtasks
 - Eventually smaller subtasks are trivial to implement in programming language
 - Very often, subtasks are implemented as functions

Function Implementations and The Black Box

- Designing with the black box in mind allows us
 - To change or improve a function definition without forcing programmers using the function to change what they have done
 - To know how to use a function simply by reading the function declaration and its comment

Benefits of Top Down Design

- Subtasks, or functions in C++, make programs easier to
 - Write: as you can focus on a simpler function, blocking out other tasks
 - Test and debug: because of simplicity of each function,
 - Understand
 - Change
 - Teamwork

Case Study Buying Pizza

- What size pizza is the best buy?
 - Which size gives the lowest cost per square inch?
 - Pizza sizes given in diameter
 - Quantity of pizza is based on the area which is proportional to the square of the radius

Buying Pizza Problem Definition

- Input:
 - Diameter of two sizes of pizza
 - Cost of the same 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

Buying Pizza Function Analysis

- Subtask 2 and subtask 3 should be implemented as a single function because
 - Subtask 2 and subtask 3 are identical tasks
 - 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
 - We'll use `unitprice`

Buying Pizza unitprice Declaration

- `double unitprice(int diameter, int 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.

Buying Pizza Algorithm Design

- Subtask 1
 - Ask for the input values and store them in variables
 - diameter_small diameter_large
 - price_small price_large
- 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 unitprice Algorithm

- Subtasks 2 and 3 are implemented as calls to function unitprice
- unitprice algorithm
 - Compute the radius of the pizza
 - Computer the area of the pizza using
 - Return the value of $(\text{price} / \text{area})$

Buying Pizza unitprice Pseudocode

- Pseudocode
 - Mixture of C++ and english
 - Allows us to make the algorithm more precise without worrying about the details of C++ syntax
- unitprice pseudocode
 - radius = one half of diameter;
area = $\pi * \text{radius} * \text{radius}$
return (price / area)

Buying Pizza The Calls of unitprice

- Main part of the program implements calls of unitprice as
 - `double unit_price_small, unit_price_large;`
`unit_price_small = unitprice(diameter_small, price_small);`
`unit_price_large = unitprice(diameter_large, price_large);`

Buying Pizza First try at unitprice

- `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 at unitprice

- 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 (part 1 of 2)

```
//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 diameter_small, diameter_large;
    double price_small, unitprice_small,
           price_large, unitprice_large;

    cout << "Welcome to the Pizza Consumers Union.\n";
    cout << "Enter diameter of a small pizza (in inches): ";
    cin >> diameter_small;
    cout << "Enter the price of a small pizza: $";
    cin >> price_small;
    cout << "Enter diameter of a large pizza (in inches): ";
    cin >> diameter_large;
    cout << "Enter the price of a large pizza: $";
    cin >> price_large;

    unitprice_small = unitprice(diameter_small, price_small);
    unitprice_large = unitprice(diameter_large, price_large);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Small pizza:\n"
           << "Diameter = " << diameter_small << " inches\n"
           << "Price = $" << price_small
           << " Per square inch = $" << unitprice_small << endl
           << "Large pizza:\n"
           << "Diameter = " << diameter_large << " inches\n"
           << "Price = $" << price_large
           << " Per square inch = $" << unitprice_large << endl;
```

```
    if (unitprice_large < unitprice_small)
        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);
}
```

Sample Dialogue

```
Welcome to the Pizza Consumers Union.
Enter diameter of a small pizza (in inches): 10
Enter the price of a small pizza: $7.50
Enter diameter of a large pizza (in inches): 13
Enter the price of a large pizza: $14.75
Small pizza:
Diameter = 10 inches
Price = $7.50 Per square inch = $0.10
Large pizza:
Diameter = 13 inches
Price = $14.75 Per square inch = $0.11
The small one is the better buy.
Buon Appetito!
```

Display 4.10 (2/2)



Overloading Example

- Revising Pizza Buying program
 - Rectangular pizzas are now offered!
 - Change input and add a function to compute unit price of a rectangular pizza
 - new function could be named `unitprice_rectangular`
 - Or, new function could be a new (overloaded) version of `unitprice` function that is already used

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

Overloading a Function Name (part 1 of 3)

```
//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 price_round, unit_price_round,  
           price_rectangular, unitprice_rectangular;  
  
    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 >> price_round;  
    cout << "Enter length and width in inches\n"  
         << "of a rectangular pizza: ";  
    cin >> length >> width;  
    cout << "Enter the price of a rectangular pizza: $";  
    cin >> price_rectangular;  
  
    unitprice_rectangular =  
        unitprice(length, width, price_rectangular);  
    unit_price_round = unitprice(diameter, price_round);  
  
    cout.setf(ios::fixed);  
    cout.setf(ios::showpoint);  
    cout.precision(2);
```

```
cout << endl
    << "Round pizza: Diameter = "
    << diameter << " inches\n"
    << "Price = $" << price_round
    << " Per square inch = $" << unit_price_round
    << endl
    << "Rectangular pizza: Length = "
    << length << " inches\n"
    << "Rectangular pizza: Width = "
    << width << " inches\n"
    << "Price = $" << price_rectangular
    << " Per square inch = $" << unitprice_rectangular
    << endl;

if (unit_price_round < unitprice_rectangular)
    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);
}
```

Overloading a Function Name (*part 3 of 3*)

Sample Dialogue

```
Welcome to the Pizza Consumers Union.  
Enter the diameter in inches of a round pizza: 10  
Enter the price of a round pizza: $8.50  
Enter length and width in inches  
of a rectangular pizza: 6 4  
Enter the price of a rectangular pizza: $7.55  
  
Round pizza: Diameter = 10 inches  
Price = $8.50 Per square inch = $0.11  
Rectangular pizza: Length = 6 inches  
Rectangular pizza: Width = 4 inches  
Price = $7.55 Per square inch = $0.31  
The round one is the better buy.  
Buon Appetito!
```

Program Testing

- Programs that compile and run can still produce errors
- Testing increases confidence that the program works correctly
 - Run the program with data that has known output
 - You may have determined this output with pencil and paper or a calculator
 - Run the program on several different sets of data
 - Your first set of data may produce correct results in spite of a logical error in the code
 - Remember the integer division problem? If there is no fractional remainder, integer division will give apparently correct results

Use Pseudocode

- Pseudocode is a mixture of English and the programming language in use
- Pseudocode simplifies algorithm design by allowing you to ignore the specific syntax of the programming language as you work out the details of the algorithm
 - If the step is obvious, use C++
 - If the step is difficult to express in C++, use English

Exercises

- Describe the purpose of the comment that accompanies a function declaration?
- Describe what it means to say a programmer should be able to treat a function as a black box?
- Describe what it means for two functions to be black box equivalent?

Exercises

- Describe Top-Down Design?
- Describe the types of tasks we have seen so far that could be implemented as C++ functions?
- Describe the principles of
 - The black box
 - Procedural abstraction
 - Information hiding
- Define “local variable”?
- Overload a function name?

Summary

- Predefined Functions
- Programmer-Defined Functions
 - Local Variables
 - Function call: how to trace?
 - Overloading Function Names
- Why functions?
 - Procedural Abstraction
 - How does it help with problem solving, i.e., Top-Down Design