2023

Program Design II

Kun-Ta Chuang
Department of Computer Science and Information Engineering
National Cheng Kung University

C++ Identifiers

- C++ identifiers 是用來表示variable、function、class、struct、enum、static value等命名實體的名稱
- 在C++中，每個identifiers都必須由以下字符組成：
  1. 英文字母（大小寫）：A-Z、a-z
  2. 數字：0-9
  3. 底線：_
  - Identifier的第一個symbol必須是一個字母或一個底線，不能是一個數字。 Identifier也不能使用關鍵字（如if、while、int等）作為其名稱。 Identifier區分大小寫
    - myVar和myvar是兩個不同的variable
    - myVariable
    - my_function
    - MyClass
    - enum_type
    - CONSTANT_VALUE
    - 1st_variable?
    - my-variable?

Data Types:
Display 1.2 Simple Types (1 of 2)

Display 1.2 Simple Types

TYPE NAME	MEMORY USED	SIZE RANGE	PRECISION
`short` (also called `short int`)	2 bytes	−32,768 to 32,767	Not applicable
`int`	4 bytes	−2,147,483,648 to 2,147,483,647	Not applicable
`long` (also called `long int`)	4 bytes	−2,147,483,648 to 2,147,483,647	Not applicable
`float`	4 bytes	approximately 10^{-38} to 10^{38}	7 digits
`double`	8 bytes	approximately 10^{-308} to 10^{308}	15 digits

Data Types:
Display 1.2 Simple Types (2 of 2)

`long double`	10 bytes	approximately 10^{-4932} to 10^{4932}	19 digits
`char`	1 byte	All ASCII characters (Can also be used as an integer type, although we do not recommend doing so.)	Not applicable
`bool`	1 byte	`true`, `false`	Not applicable

The values listed here are only sample values to give you a general idea of how the types differ. The values for any of these entries may be different on your system. *Precision* refers to the number of meaningful digits, including digits in front of the decimal point. The ranges for the types `float`, `double`, and `long double` are the ranges for positive numbers. Negative numbers have a similar range, but with a negative sign in front of each number.

C++11 Fixed Width Integer Types

TYPE NAME	MEMORY USED	SIZE RANGE
int8_t	1 byte	–128 to 127
uint8_t	1 byte	0 to 255
int16_t	2 bytes	–32,768 to 32,767
uint16_t	2 bytes	0 to 65,535
int32_t	4 bytes	–2,147,483,648 to 2,147,483,647
uint32_t	4 bytes	0 to 4,294,967,295
int64_t	8 bytes	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
uint64_t	8 bytes	0 to 18,446,744,073,709,551,615
long long	At least 8 bytes	

Avoids problem of variable integer sizes for different CPUs
這些類型可以用來替代C++中的int、short、long、long long等類型以確保在不同平台上整數類型具有相同的大小和行為。

Assigning Data: Shorthand Notations

EXAMPLE	EQUIVALENT TO
`count += 2;`	`count = count + 2;`
`total -= discount;`	`total = total - discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rushFactor;`	`time = time/rushFactor;`
`change %= 100;`	`change = change % 100;`
`amount *= cnt1 + cnt2;`	`amount = amount * (cnt1 + cnt2);`

Data Assignment Rules

- Compatibility of Data Assignments
  - Type mismatches
    - General Rule: Cannot place value of one type into variable of another type
  - intVar = 2.99; // 2 is assigned to intVar!
    - Only integer part "fits"
    - Called "implicit" or "automatic type conversion"
  - Literals
    - 2, 5.75, "Z", "Hello World"
    - Considered "constants": can’t change in program
      - const int MAX_VALUE = 100;
      - const string GREETING_MESSAGE = "Hello, World!";

Arithmetic Precision

- Precision of Calculations
  - VERY important consideration!
    - Expressions in C++ might not evaluate as you'd "expect"!
  - "Highest-order operand" determines type of arithmetic "precision" performed
  - Common pitfall!

Arithmetic Precision Examples

- Examples:
  - 17 / 5 evaluates to 3 in C++!
    - Both operands are integers
    - Integer division is performed!
  - 17.0 / 5 equals 3.4 in C++!
    - Highest-order operand is "double type"
    - Double "precision" division is performed!
  - int intVar1 =1, intVar2=2;
    intVar1 / intVar2;
    - Performs integer division!
    - Result: 0!

Individual Arithmetic Precision

- Calculations done "one-by-one"
  - 1 / 2 / 3.0 / 4 performs 3 separate divisions.
    - First→ 1 / 2   equals 0
    - Then→ 0 / 3.0 equals 0.0
    - Then→ 0.0 / 4 equals 0.0!
- So not necessarily sufficient to change just "one operand" in a large expression
  - Must keep in mind all individual calculations that will be performed during evaluation!

Type Casting

- Two types
  - Implicit—also called "Automatic"
    - Done FOR you, automatically
      17 / 5.5
      This expression causes an "implicit type cast" to take place, casting the 17 → 17.0
  - Explicit type conversion
    - Programmer specifies conversion with cast operator
      (double)17 / 5.5
      Same expression as above, using explicit cast
      (double)myInt / myDouble
      More typical use; cast operator on variable

Shorthand Operators

- Increment & Decrement Operators
  - Just short-hand notation
  - Increment operator, ++
    intVar++; is equivalent to
    intVar = intVar + 1;
  - Decrement operator, --
    intVar--; is equivalent to
    intVar = intVar – 1;

Shorthand Operators: Two Options

- Post-Increment
  intVar++
  - Uses current value of variable, THEN increments it
- Pre-Increment
  ++intVar
  - Increments variable first, THEN uses new value
- No difference if "alone" in statement:
  intVar++; and ++intVar; → identical result

Post-Increment in Action

- Post-Increment in Expressions:

```
int           n = 2,
              valueProduced;
valueProduced = 2 * (n++);
cout << valueProduced << endl;
cout << n << endl;
```

  - This code segment produces the output:
    4
    3
  - Since post-increment was used

Pre-Increment in Action

- Now using Pre-increment:

```
int           n = 2,
              valueProduced;
valueProduced = 2 * (++n);
cout << valueProduced << endl;
cout << n << endl;
```

  - This code segment produces the output:
    6
    3
  - Because pre-increment was used

String type

- C++ has a data type of "string" to store sequences of characters
  - Not a primitive data type
  - Must add `#include <string>` at the top of the program
  - The "+" operator on strings concatenates two strings together
  - cin >> str where str is a string only reads up to the first whitespace character

Input/Output (1 of 2)

Display 1.5 Using `cin` and `cout` with a string (part 1 of 2)

```
1  //Program to demonstrate cin and cout with strings
2  #include <iostream>
3  #include <string>

4  using namespace std;
5  int main( )
6  {
7    string dogName;
8    int actualAge;
9    int humanAge;

10   cout << "How many years old is your dog?" << endl;
11   cin >> actualAge;
12   humanAge = actualAge * 7;

13   cout << "What is your dog's name?" << endl;
14   cin >> dogName;

15   cout << dogName << "'s age is approximately " <<
16          "equivalent to a " << humanAge << " year old human."
17          << endl;

18   return 0;
19 }
```

Needed to access the `string` *class.*

Input/Output (2 of 2)

Display 1.5 Using `cin` and `cout` with a string (part 2 of 2)

Sample Dialogue 1

```
How many years old is your dog?
5
What is your dog's name?
Rex
Rex's age is approximately equivalent to a 35 year old human.
```

Sample Dialogue 2

```
How many years old is your dog?
10
What is your dog's name?
Mr. Bojangles
Mr.'s age is approximately equivalent to a 70 year old human.
```

"Bojangles" is not read into `dogName` *because* `cin` *stops input at the space.*

Recap

- C++ is case-sensitive
- Use meaningful names
  - For variables and constants
- Variables should be declared before use
  - Should also be initialized
- Use care in numeric manipulation
  - Precision, parentheses, order of operations

Precedence Examples

- Arithmetic before logical
  - x + 1 > 2 || x + 1 < -3 means:
    - (x + 1) > 2 || (x + 1) < -3
- Short-circuit evaluation
  - (x >= 0) && (y > 1)
  - Be careful with increment operators!
    - (x > 1) && (y++)
- Integers as boolean values
  - All non-zero values → true
  - Zero value → false

Common Pitfalls

- Operator "=" vs. operator "=="
- One means "assignment" (=)
- One means "equality" (==)
  - VERY different in C++!
  - Example:

```
if (x = 12)  ←Note operator used!
    Do_Something
else
    Do_Something_Else
```

The switch Statement

- A statement for controlling multiple branches
- Can do the same thing with if statements but sometimes switch is more convenient
- Uses controlling expression which returns bool data type (true or false)

switch Statement Syntax

switch Statement

SYNTAX

```
switch (Controlling_Expression)
{
    case Constant_1:
        Statement_Sequence_1
        break;
    case Constant_2:
        Statement_Sequence_2
        break;
                    .
                    .
                    .
    case Constant_n:
        Statement_Sequence_n
        break;
    default:
        Default_Statement_Sequence
}
```

You need not place a `break` *statement in each case. If you omit a* `break`*, that case continues until a* `break` *(or the end of the* `switch` *statement) is reached.*

The controlling expression must be integral! This includes char.

The switch Statement in Action

EXAMPLE

```
int vehicleClass;
double toll;
cout << "Enter vehicle class: ";
cin >> vehicleClass;

switch (vehicleClass)
{
    case 1:
        cout << "Passenger car.";
        toll = 0.50;
        break;
    case 2:
        cout << "Bus.";
        toll = 1.50;
        break;
    case 3:
        cout << "Truck.";
        toll = 2.00;
        break;
    default:
        cout << "Unknown vehicle class!";
}
```

If you forget this `break`*, then passenger cars will pay $1.50.*

Conditional Operator

- Also called "ternary operator"
  - Allows embedded conditional in expression
  - Essentially "shorthand if-else" operator
  - Example:
    ```
    if (n1 > n2)
        max = n1;
    else
        max = n2;
    ```
  - Can be written:
    ```
    max = (n1 > n2) ? N1 : n2;
    ```
    - "?" and ":" form this "ternary" operator

Loops

- 3 Types of loops in C++
  - while
    - Most flexible
    - No "restrictions"
  - do-while
    - Least flexible
    - Always executes loop body at least once
  - for
    - Natural "counting" loop

while Loops Syntax

Syntax for `while` and `do-while` Statements

A `while` STATEMENT WITH A SINGLE STATEMENT BODY

```
while (Boolean_Expression)
     Statement
```

A `while` STATEMENT WITH A MULTISTATEMENT BODY

```
while (Boolean_Expression)
{
     Statement_1
     Statement_2
        .
        .
        .
     Statement_Last
}
```

while Loop Example

- Consider:

```
count = 0;                  // Initialization
while (count < 3)           // Loop Condition
{
    cout << "Hi ";          // Loop Body
    count++;                // Update expression
}
```

  - Loop body executes how many times?

do-while Loop Syntax

A do-while STATEMENT WITH A SINGLE-STATEMENT BODY

```
do
    Statement
while (Boolean_Expression);
```

A do-while STATEMENT WITH A MULTISTATEMENT BODY

```
do
{
    Statement_1
    Statement_2
      .
      .
      .
    Statement_Last
} while (Boolean_Expression);
```

Do not forget the final semicolon.

do-while Loop Example

- ```
  count = 0;                  // Initialization
  do
  {
      cout << "Hi ";          // Loop Body
      count++;                // Update expression
  } while (count < 3);        // Loop Condition
  ```
  - Loop body executes how many times?
  - do-while loops always execute body at least once!
  ```

while vs. do-while

- Very similar, but…
  - One important difference
    - Issue is "WHEN" boolean expression is checked
    - while: checks BEFORE body is executed
    - do-while: checked AFTER body is executed
- After this difference, they're essentially identical!
- while is more common, due to it's ultimate "flexibility"

Comma Operator

- Evaluate list of expressions, returning value of the last expression
- Most often used in a for-loop
- Example:
  first = (first = 2, second = first + 1);
  - first gets assigned the value 3
  - second gets assigned the value 3
- No guarantee what order expressions will be evaluated.

for Loop Syntax

for (Init_Action; Bool_Exp; Update_Action)
    Body_Statement

- Like if-else, Body_Statement can be a block statement
  - Much more typical

for Loop Example

- for (count=0;count<3;count++)
  {
  cout << "Hi "; // Loop Body
  }
- How many times does loop body execute?
- Initialization, loop condition and update all "built into" the for-loop structure!
- A natural "counting" loop

Loop Issues

- Loop's condition expression can be ANY boolean expression
- Examples:

```
while (count<3 && done!=0)
{
    // Do something
}
for (index=0;index<10 && entry!=-99;)
{
    // Do something
}
```

Loop Pitfalls: Misplaced ;

- Watch the misplaced ; (semicolon)
  - Example:

```
while (response != 0) ;←
{
    cout << "Enter val: ";
    cin >> response;
}
```

  - Notice the ";" after the while condition!
- Result here: INFINITE LOOP!

Loop Pitfalls: Infinite Loops

- Loop condition must evaluate to false at some iteration through loop
  - If not → infinite loop.
  - Example:

```
while (1)
{
    cout << "Hello ";
}
```

  - A perfectly legal C++ loop → always infinite!
- Infinite loops can be desirable
  - e.g., "Embedded Systems“ or Internet Services

The break and continue Statements

- Flow of Control
  - Recall how loops provide "graceful" and clear flow of control in and out
  - In RARE instances, can alter natural flow
- break;
  - Forces loop to exit immediately.
- continue;
  - Skips rest of loop body
- These statements violate natural flow
  - Only used when absolutely necessary!

Nested Loops

- Recall: ANY valid C++ statements can be inside body of loop
- This includes additional loop statements!
  - Called "nested loops"
- Requires careful indenting:

```
for (outer=0; outer<5; outer++)
    for (inner=7; inner>2; inner--)
        cout << outer << inner;
```

  - Notice no { } since each body is one statement
  - Good style dictates we use { } anyway

Programmer-Defined Functions

- Write your own functions!
- Building blocks of programs
  - Divide & Conquer
  - Readability
  - Re-use
- Your "definition" can go in either:
  - Same file as main()
  - Separate file so others can use it, too

Components of Function Use

- 3 Pieces to using functions:
  - Function Declaration/prototype
    - Information for compiler
    - To properly interpret calls
  - Function Definition
    - Actual implementation/code for what function does
  - Function Call
    - Transfer control to function

Function Declaration

- Also called function prototoype
- An "informational" declaration for compiler
- Tells compiler how to interpret calls
  - Syntax:
    <return_type> FnName(<formal-parameter-list>);
  - Example:
    double totalCost(int numberParameter,
    double priceParameter);
- Placed before any calls
  - In declaration space of main()
  - Or above main() in global space

Function Definition

- Implementation of function
- Just like implementing function main()
- Example:

```
double totalCost(          int numberParameter,
                           double priceParameter)
{
      const double TAXRATE = 0.05;
      double subTotal;
      subtotal = priceParameter * numberParameter;
      return (subtotal + subtotal * TAXRATE);
}
```

- Notice proper indenting

Function Definition Placement

- Placed after function main()
  - NOT "inside" function main()!
- Functions are "equals"; no function is ever "part" of another
- Formal parameters in definition
  - "Placeholders" for data sent in
    - "Variable name" used to refer to data in definition
- return statement
  - Sends data back to caller

Function Call

- Just like calling predefined function
  bill = totalCost(number, price);
  - totalCost returns double value
  - Assigned to variable named "bill"
- Arguments here: number, price
  - Recall arguments can be literals, variables, expressions, or combination
  - In function call, arguments often called "actual arguments"
    - Because they contain the "actual data" being sent

Function Example: A Function to Calculate Total Cost (1 of 2)

Display 3.5

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

3   double totalCost(int numberParameter, double priceParameter);
4   //Computes the total cost, including 5% sales tax,
5   //on numberParameter items at a cost of priceParameter each.

6   int main( )
7   {
8       double price, bill;
9       int number;

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

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

Function declaration; also called the function prototype

Function call

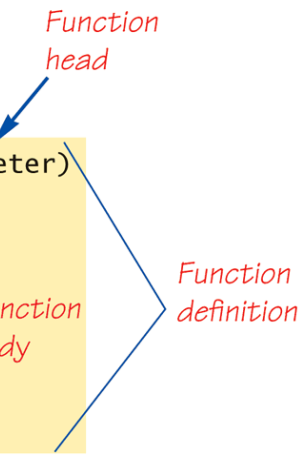
Function Example: A Function to Calculate Total Cost (1 of 2)

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

22      return 0;
23  }

24  double totalCost(int numberParameter, double priceParameter)
25  {
26      const double TAXRATE = 0.05; //5% sales tax
27      double subtotal;

28      subtotal = priceParameter * numberParameter;
29      return (subtotal + subtotal*TAXRATE);
30  }
```



SAMPLE DIALOGUE

Enter the number of items purchased: **2**
Enter the price per item: $**10.10**
2 items at $10.10 each.
Final bill, including tax, is $21.21

Alternative Function Declaration

- Recall: Function declaration is "information" for compiler
- Compiler only needs to know:
  - Return type
  - Function name
  - Parameter list
- Formal parameter names not needed:
  double totalCost(int, double);
  - Still "should" put in formal parameter names
    - Improves readability

Parameter vs. Argument

- Terms often used interchangeably
- Formal parameters/arguments
  - In function declaration
  - In function definition's header
- Actual parameters/arguments
  - In function call
- Technically parameter is "formal" piece while argument is "actual" piece*
  - *Terms not always used this way

Functions Calling Functions

- We’re already doing this!
  - main() is a function!
- Only requirement:
  - Function’s declaration must appear first
- Function’s definition typically elsewhere
  - After main() definition
  - Or in separate file
- Common for functions to call many other functions
- Function can even call itself → "Recursion"

main(): "Special"

- Recall: main() IS a function
- "Special" in that:
  - One and only one function called main() will exist in a program
- Who calls main()?
  - Operating system
  - Tradition holds it should have return statement
    - Value returned to "caller" → Here: operating system
  - Should return "int" or "void"

Parameters and Overloading

Learning Objectives

- Parameters
  - Call-by-value
  - Call-by-reference
  - Mixed parameter-lists
- Overloading and Default Arguments
  - Examples, Rules
- Testing and Debugging Functions
  - assert Macro
  - Stubs, Drivers

Parameters

- Two methods of passing arguments as parameters
- Call-by-value
  - "copy" of value is passed
- Call-by-reference
  - "address of" actual argument is passed

Call-by-Value Parameters

- Copy of actual argument passed
- Considered "local variable" inside function
- If modified, only "local copy" changes
  - Function has no access to "actual argument" from caller
- This is the default method
  - Used in all examples thus far

Call-by-Value Example: Formal Parameter Used as a Local Variable (1 of 3)

Display 4.1 Formal Parameter Used as a Local Variable

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

4   const double RATE = 150.00; //Dollars per quarter hour.
                        try remove          try remove
5   double fee(int hoursWorked, int minutesWorked);
6   //Returns the charges for hoursWorked hours and
7   //minutesWorked minutes of legal services.

8   int main( )
9   {
10      int hours, minutes;
11      double bill;
```

Call-by-Value Example: Formal Parameter Used as a Local Variable (2 of 3)

```
12      cout << "Welcome to the law office of\n"
13           << "Dewey, Cheatham, and Howe.\n"
14           << "The law office with a heart.\n"
15           << "Enter the hours and minutes"
16           << " of your consultation:\n";
17      cin >> hours >> minutes;

18      bill = fee(hours, minutes);

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

24      return 0;
25  }
```

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

(continued)

Call-by-Value Example: Formal Parameter Used as a Local Variable (3 of 3)

Display 4.1 Formal Parameter Used as a Local Variable

```
26  double fee(int hoursWorked, int minutesWorked)
27  {
28      int quarterHours;

29      minutesWorked = hoursWorked*60 + minutesWorked;
30      quarterHours = minutesWorked/15;
31      return (quarterHours*RATE);
32  }
```

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

SAMPLE DIALOGUE

```
Welcome to the law office of
Dewey, Cheatham, and Howe.
The law office with a heart.
Enter the hours and minutes of your consultation:
5 46
For 5 hours and 46 minutes, your bill is $3450.00
```

Call-by-Value Pitfall

- Common Mistake:
  - Declaring parameter "again" inside function:

```
double fee(int hoursWorked, int minutesWorked)
{
  int quarterHours;          // local variable
  int minutesWorked          // NO!
}
```

  - Compiler error results
    - "Redefinition error…"
- Value arguments ARE like "local variables"
  - But function gets them "automatically"

Call-By-Reference Parameters

- Used to provide access to caller's actual argument
- Caller's data can be modified by called function!
- Typically used for input function
  - To retrieve data for caller
  - Data is then "given" to caller
- Specified by ampersand, &, after type in formal parameter list

Call-By-Reference Example: Call-by-Reference Parameters (1 of 3)

Display 4.2 Call-by-Reference Parameters

```
1  //Program to demonstrate call-by-reference parameters.
2  #include <iostream>
3  using namespace std;

4  void getNumbers(int& input1, int& input2);
5  //Reads two integers from the keyboard.

6  void swapValues(int& variable1, int& variable2);
7  //Interchanges the values of variable1 and variable2.

8  void showResults(int output1, int output2);
9  //Shows the values of variable1 and variable2, in that order.

10 int main( )
11 {
12     int firstNum, secondNum;

13     getNumbers(firstNum, secondNum);
14     swapValues(firstNum, secondNum);
15     showResults(firstNum, secondNum);
16     return 0;
17 }
```

Call-By-Reference Example: Call-by-Reference Parameters (2 of 3)

```
18  void getNumbers(int& input1, int& input2)
19  {
20      cout << "Enter two integers: ";
21      cin >> input1
22          >> input2;
23  }

24  void swapValues(int& variable1, int& variable2)
25  {
26      int temp;

27      temp = variable1;
28      variable1 = variable2;
29      variable2 = temp;
30  }
31
32  void showResults(int output1, int output2)
33  {
34      cout << "In reverse order the numbers are: "
35           << output1 << " " << output2 << endl;
36  }
```

Call-By-Reference Example: Call-by-Reference Parameters (3 of 3)

Display 4.2 Call-by-Reference Parameters

SAMPLE DIALOGUE

```
Enter two integers: 5 6
In reverse order the numbers are: 6 5
```

Call-By-Reference Details

- What’s really passed in?
- A "reference" back to caller’s actual argument!
  - Refers to memory location of actual argument
  - Called "address", which is a unique number referring to distinct place in memory

Constant Reference Parameters

- Reference arguments inherently "dangerous"
  - Caller's data can be changed
  - Often this is desired, sometimes not
- To "protect" data, & still pass by reference:
  - Use const keyword
    - void sendConstRef(const int &par1,
      const int &par2);
    - Makes arguments "read-only" by function
    - No changes allowed inside function body

Difference between CbA and CbR

- 在C++中，Call-by-Address和Call-by-Reference都是用於傳遞函數參數的方法，它們有一些相似之處，但也有一些重要的差別。
- Call-by-Address是通過將變量的內存地址傳遞給函數來傳遞變量。在函數中，通過該地址可以訪問和修改原始變量的值。在使用Call-by-Address時，函數的參數是指針類型，而在函數內部，我們必須通過指針間接訪問和修改變量的值。
- Call-by-Reference是通過將變量的引用傳遞給函數來傳遞變量。在函數中，通過引用可以直接訪問和修改原始變量的值。在使用Call-by-Reference時，函數的參數是引用類型，而在函數內部，我們可以像使用普通變量一樣直接訪問和修改引用所對應的變量的值。

Difference between CbA and CbR

- 因此，主要的差異在於傳遞參數的方式和在函數中訪問和修改變量的方式。使用Call-by-Reference比使用Call-by-Address更加簡單，因為它不需要通過間接訪問指針來訪問和修改變量的值。另外，使用Call-by-Reference也更加安全，因為它可以避免指針操作時可能產生的錯誤。

- 需要注意的是，儘管在C++中使用了Call-by-Reference，但實際上它是通過指針實現的。因此，在使用Call-by-Reference時，我們可以將它視為Call-by-Address的一種簡化寫法。

Call-by-Address

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

void swap(int* x, int* y) {  // 接受兩個int類型的指針
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main() {
    int a = 10, b = 20;
    int* pa = &a;
    int* pb = &b;
    cout << "Before swap: a = " << a << ", b = " << b << endl;
    swap(pa, pb);
    cout << "After swap: a = " << a << ", b = " << b << endl;
    return 0;
}
```

Call-by-Reference

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

void swap(int& x, int& y) {  // 接受兩個int類型的參考
    int temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 10, b = 20;
    cout << "Before swap: a = " << a << ", b = " << b << endl;
    swap(a, b);
    cout << "After swap: a = " << a << ", b = " << b << endl;
    return 0;
}
```

Mixed Parameter Lists

- Can combine passing mechanisms
- Parameter lists can include pass-by-value and pass-by-reference parameters
- Order of arguments in list is critical:
  void mixedCall(int & par1, int par2, double & par3);
  - Function call:
    mixedCall(arg1, arg2, arg3);
    - arg1 must be integer type, is passed by reference
    - arg2 must be integer type, is passed by value
    - arg3 must be double type, is passed by reference

Overloading

- Same function name
- Different parameter lists
- Two separate function definitions
- Function "signature"
  - Function name & parameter list
  - Must be "unique" for each function definition
- Allows same task performed on different data

Overloading Example: Average

- Function computes average of 2 numbers:

```
double average(double n1, double n2)
{
        return ((n1 + n2) / 2.0);
}
```

- Now compute average of 3 numbers:

```
double average(double n1, double n2, double n3)
{
        return ((n1 + n2) / 2.0);
}
```

- Same name, two functions

Overloaded Average() Cont'd

- Which function gets called?
- Depends on function call itself:
  - avg = average(5.2, 6.7);
    - Calls "two-parameter average()"
  - avg = average(6.5, 8.5, 4.2);
    - Calls "three-parameter average()"
- Compiler resolves invocation based on signature of function call
  - "Matches" call with appropriate function
  - Each considered separate function

Overloading Pitfall

- Only overload "same-task" functions
  - A mpg() function should always perform same task, in all overloads
  - Otherwise, unpredictable results
- C++ function call resolution:
  - 1st: looks for exact signature
  - 2nd: looks for "compatible" signature

Overloading Resolution

- 1st: Exact Match
  - Looks for exact signature
    - Where no argument conversion required
- 2nd: Compatible Match
  - Looks for "compatible" signature where automatic type conversion is possible:
    - 1st with promotion (e.g., int→double)
      - No loss of data
    - 2nd with demotion (e.g., double→int)
      - Possible loss of data

Overloading Resolution Example

- Given following functions:
  - 1. void f(int n, double m);
    2. void f(double n, int m);
    3. void f(int n, int m);
  - These calls:
    f(98, 99); → Calls #3
    f(5.3, 4); → Calls #2
    f(4.3, 5.2); → Calls ???
- Avoid such confusing overloading

Automatic Type Conversion and Overloading

- Numeric formal parameters typically made "double" type
- Allows for "any" numeric type
  - Any "subordinate" data automatically promoted
    - int → double
    - float → double
    - char → double *More on this later!
- Avoids overloading for different numeric types

Automatic Type Conversion and Overloading Example

- double mpg(double miles, double gallons)
  {
  return (miles/gallons);
  }
- Example function calls:
  - mpgComputed = mpg(5, 20);
    - Converts 5 & 20 to doubles, then passes
  - mpgComputed = mpg(5.8, 20.2);
    - No conversion necessary
  - mpgComputed = mpg(5, 2.4);
    - Converts 5 to 5.0, then passes values to function

Default Arguments

- Allows omitting some arguments
- Specified in function declaration/prototype
  - void showVolume(int length,
    int width = 1,
    int height = 1);
    - Last 2 arguments are defaulted
  - Possible calls:
    - showVolume(2, 4, 6); //All arguments supplied
    - showVolume(3, 5); //height defaulted to 1
    - showVolume(7); //width & height defaulted to 1

Default Arguments Example: Default Arguments (1 of 2)

Display 4.8 Default Arguments

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

4   void showVolume(int length, int width = 1, int height = 1);
5   //Returns the volume of a box.
6   //If no height is given, the height is assumed to be 1.
7   //If neither height nor width is given, both are assumed to be 1.

8   int main( )
9   {
10      showVolume(4, 6, 2);
11      showVolume(4, 6);
12      showVolume(4);

13      return 0;
14  }

15  void showVolume(int length, int width, int height)
```

Default arguments

A default argument should not be given a second time.

Default Arguments Example: Default Arguments (2 of 2)

```
16  {
17      cout << "Volume of a box with \n"
18           << "Length = " << length << ", Width = " << width << endl
19           << "and Height = " << height
20           << " is " << length*width*height << endl;
21  }
```

SAMPLE DIALOGUE

```
Volume of a box with
Length = 4, Width = 6
and Height = 2 is 48
Volume of a box with
Length = 4, Width = 6
and Height = 1 is 24
Volume of a box with
Length = 4, Width = 1
and Height = 1 is 4
```

Testing and Debugging Functions

- Many methods:
  - Lots of cout statements
    - In calls and definitions
    - Used to "trace" execution
  - Compiler Debugger
    - Environment-dependent
  - assert Macro
    - Early termination as needed
  - Stubs and drivers
    - Incremental development

The assert Macro

- Assertion: a true or false statement
- Used to document and check correctness
  - Preconditions & Postconditions
    - Typical assert use: confirm their validity
  - Syntax:
    assert(<assert_condition>);
    - No return value
    - Evaluates assert_condition
    - Terminates if false, continues if true
- Predefined in library <cassert>
  - Macros used similarly as functions

An assert Macro Example

- Given Function Declaration:

```
void computeCoin(        int coinValue,
                         int& number,
                         int& amountLeft);
//Precondition: 0 < coinValue < 100
                        0 <= amountLeft <100
//Postcondition: number set to max. number
                          of coins
```

- Check precondition:
  - assert ((0 < currentCoin) && (currentCoin < 100) && (0 <= currentAmountLeft) && (currentAmountLeft < 100));
  - If precondition not satisfied → condition is false → program execution terminates!

An assert Macro Example Cont'd

- Useful in debugging
- Stops execution so problem can be investigated

assert On/Off

- Preprocessor provides means
- #define NDEBUG
  #include <cassert>
- Add "#define" line before #include line
  - Turns OFF all assertions throughout program
- Remove "#define" line (or comment out)
  - Turns assertions back on

Stubs and Drivers

- Separate compilation units
  - Each function designed, coded, tested separately
  - Ensures validity of each unit
  - Divide & Conquer
    - Transforms one big task → smaller, manageable tasks
- But how to test independently?
  - Driver programs

Stubs and Drivers

- Stubs和Drivers是軟件測試中常用的概念，用於測試不同部分之間的集成和交互。
- Stubs是一種用於測試某個模塊的軟件元素，通常是一個假的模塊或者模塊的部分，用於模擬一個真實的模塊或者模塊的部分。Stubs可以在模擬的模塊或者模塊的部分還沒有完成時就可以測試其他部分的代碼。通常，Stubs用於代替還未完成或者還未可用的代碼，以便在測試時能夠快速地進行集成和測試。Stubs通常只實現了被測試代碼所需的最基本的功能。
- Drivers是另一種軟件元素，用於測試某個模塊或者系統的部分。它通常是一個輔助的代碼模塊，用於調用被測試模塊的特定函數或者方法，以便在測試時能夠模擬對被測試代碼的實際調用。Driver通常用於測試已經完成的模塊或者系統的部分，以便確定其功能是否正常運行。
- 簡而言之，Stubs和Drivers是用於軟件測試中的工具，Stubs通常用於模擬還未完成或者還未可用的代碼，Drivers則用於調用已經完成的代碼的特定函數或者方法，以便在測試時能夠模擬對被測試代碼的實際調用。使用Stubs和Drivers可以幫助測試人員快速地進行集成和測試，以確保整個系統的正常運行。

Driver Program Example: Driver Program (1 of 3)

Display 4.9 Driver Program

```
1
2   //Driver program for the function unitPrice.
3   #include <iostream>
4   using namespace std;

5   double unitPrice(int diameter, double price);
6   //Returns the price per square inch of a pizza.
7   //Precondition: The diameter parameter is the diameter of the pizza
8   //in inches. The price parameter is the price of the pizza.

9   int main( )
10  {
11      double diameter, price;
12      char ans;

13      do
14      {
15          cout << "Enter diameter and price:\n";
16          cin >> diameter >> price;
```

Driver Program Example: Driver Program (2 of 3)

```
17          cout << "unit Price is $"
18               << unitPrice(diameter, price) << endl;

19          cout << "Test again? (y/n)";
20          cin >> ans;
21          cout << endl;
22      } while (ans == 'y' || ans == 'Y');

23      return 0;
24  }
25
26  double unitPrice(int diameter, double price)
27  {
28      const double PI = 3.14159;
29      double radius, area;

30      radius = diameter/static_cast<double>(2);
31      area = PI * radius * radius;
32      return (price/area);
33  }
```

(continued)

Driver Program Example: Driver Program (3 of 3)

Display 4.9 Driver Program

SAMPLE DIALOGUE

```
Enter diameter and price:
13 14.75
Unit price is: $0.111126
Test again? (y/n): y

Enter diameter and price:
2 3.15
Unit price is: $1.00268
Test again? (y/n): n
```

Stubs

- Develop incrementally
- Write "big-picture" functions first
  - Low-level functions last
  - "Stub-out" functions until implementation
  - Example:

```
double unitPrice(int diameter, double price)
{
    return (9.99);        // not valid, but noticeably
                          // a "temporary" value
}
```

  - Calls to function will still "work"

Fundamental Testing Rule

- To write "correct" programs
- Minimize errors, "bugs"
- Ensure validity of data
  - Test every function in a program where every other function has already been fully tested and debugged
  - Avoids "error-cascading" & conflicting results