2023

Program Design II

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

Learning Objectives

- Basic Operator Overloading
  - Unary operators
  - As member functions
- Friends and Automatic Type Conversion
  - Friend functions, friend classes
  - Constructors for automatic type conversion
- References and More Overloading
  - << and >>
  - Operators: = , [], ++, --

Operator Overloading Introduction

- Operators +, -, %, ==, etc.
  - Really just functions!
- Simply "called" with different syntax:
  x + 7
  - "+" is binary operator with x & 7 as operands
  - We "like" this notation as humans
- Think of it as:
  +(x, 7)
  - "+" is the function name
  - x, 7 are the arguments
  - Function "+" returns "sum" of it’s arguments

Operator Overloading Perspective

- Built-in operators
  - e.g., +, -, = , %, ==, /, *
  - Already work for C++ built-in types
  - In standard "binary" notation
- We can overload them!
  - To work with OUR types!
  - To add "Chair types", or "Money types"
    - As appropriate for our needs
    - In "notation" we're comfortable with
- Always overload with similar "actions"!

Overloading Basics

- Overloading operators
  - VERY similar to overloading functions
  - Operator itself is "name" of function
- Example Declaration:

```
const Money operator +(      const Money& amount1,
                             const Money& amount2);
```

  - Overloads + for operands of type Money
  - Uses constant reference parameters for efficiency
  - Returned value is type Money
    - Allows addition of "Money" objects

Overloaded "+"

- Given previous example:
  - Note: overloaded "+" NOT member function
  - Definition is "more involved" than simple "add"
    - Requires issues of money type addition
    - Must handle negative/positive values
- Operator overload definitions generally very simple
  - Just perform "addition" particular to "your" type

Money "+" Definition:
Display 8.1 Operator Overloading

- Definition of "+" operator for Money class:

```
const Money operator +(const Money& amount1, const Money& amount2)
{
    int allCents1 = amount1.getCents( ) + amount1.getDollars( )*100;
    int allCents2 = amount2.getCents( ) + amount2.getDollars( )*100;
    int sumAllCents = allCents1 + allCents2;
    int absAllCents = abs(sumAllCents); //Money can be negative.
    int finalDollars = absAllCents/100;
    int finalCents = absAllCents%100;

    if (sumAllCents < 0)
    {
        finalDollars = -finalDollars;
        finalCents = -finalCents;
    }

    return Money(finalDollars, finalCents);
}
```

If the **return** *statements puzzle you, see the tip entitled* ***A Constructor Can Return an Object.***

Overloaded "=="

- Equality operator, ==
  - Enables comparison of Money objects
  - Declaration:
    bool operator ==(const Money& amount1,
    const Money& amount2);
    - Returns bool type for true/false equality
  - Again, it's a non-member function
    (like "+" overload)

Overloaded "==" for Money: **Display 8.1** Operator Overloading

- Definition of "==" operator for Money class:

```
bool operator ==(const Money& amount1, const Money& amount2)
{
    return ((amount1.getDollars( ) == amount2.getDollars( ))
           && (amount1.getCents( ) == amount2.getCents( )));
}
```

Constructors Returning Objects

- Constructor a "void" function?
  - We "think" that way, but no
  - A "special" function
    - With special properties
    - CAN return a value!
- Recall return statement in "+" overload for Money type:
  - return Money(finalDollars, finalCents);
    - Returns an "invocation" of Money class!
    - So constructor actually "returns" an object!
    - Called an "anonymous object"

Returning by const Value

- Consider "+" operator overload again:

```
const Money operator +(const Money& amount1,
                       const Money& amount2);
```

  - Returns a "constant object"?
  - Why?
- Consider impact of returning "non-const" object to see…→

Returning by non-const Value

- Consider "no const" in declaration:
  Money operator +(const Money& amount1,
  const Money& amount2);
- Consider expression that calls:
  m1 + m2
  - Where m1 & m2 are Money objects
  - Object returned is Money object
  - We can "do things" with objects!
    - Like call member functions…

What to do with Non-const Object

- Can call member functions:
  - We could invoke member functions on object returned by expression m1+m2:
    - (m1+m2).output(); //Legal, right?
      - Not a problem: doesn't change anything
    - (m1+m2).input(); //Legal!
      - PROBLEM! //Legal, but MODIFIES!
    - Allows modification of "anonymous" object!
    - Can't allow that here!
- So we define the return object as const

Overloading Unary Operators

- C++ has unary operators:
  - Defined as taking one operand
  - e.g., - (negation)
    - x = -y; // Sets x equal to negative of y
  - Other unary operators:
    - ++, --
- Unary operators can also be overloaded

Overload "-" for Money

- Overloaded "-" function declaration
  - Placed outside class definition:
    const Money operator –(const Money& amount);
  - Notice: only one argument
    - Since only 1 operand (unary)
- "-" operator is overloaded twice!
  - For two operands/arguments (binary)
  - For one operand/argument (unary)
  - Definitions must exist for both

Overloaded "-" Definition

- Overloaded "-" function definition:

```
const Money operator –(const Money& amount)
{
      return Money(-amount.getDollars(),
                   -amount.getCents());
}
```

- Applies "-" unary operator to built-in type
  - Operation is "known" for built-in types
- Returns anonymous object again

Overloaded "-" Usage

- Consider:

```
Money    amount1(10),
         amount2(6),
         amount3;
amount3 = amount1 – amount2;
```

  - Calls binary "-" overload

```
amount3.output();     //Displays $4.00
amount3 = -amount1;
```

  - Calls unary "-" overload

```
amount3.output()      //Displays -$10.00
```

Overloading as Member Functions

- Previous examples: standalone functions
  - Defined outside a class
- Can overload as "member operator"
  - Considered "member function" like others
- When operator is member function:
  - Only ONE parameter, not two!
  - Calling object serves as 1st parameter

Member Operator in Action

- Money cost(1, 50), tax(0, 15), total;
  total = cost + tax;
  - If "+" overloaded as member operator:
    - Variable/object cost is calling object
    - Object tax is single argument
  - Think of as: total = cost.+(tax);
- Declaration of "+" in class definition:
  - const Money operator +(const Money& amount);
  - Notice only ONE argument

const Functions

- When to make function const?
  - Constant functions not allowed to alter class member data
  - Constant objects can ONLY call constant member functions
- Good style dictates:
  - Any member function that will NOT modify data should be made const
- Use keyword *const* after function declaration and heading

Overloading Operators: Which Method?

- Object-Oriented-Programming
  - Principles suggest member operators
  - Many agree, to maintain "spirit" of OOP
- Member operators more efficient
  - No need to call accessor & mutator functions
- At least one significant disadvantage
  - (Later in chapter…)

Overloading Function Application ()

- Function call operator, ( )
  - Must be overloaded as member function
  - Allows use of class object like a function
  - Can overload for all possible numbers of arguments
- Example:

```
Aclass anObject;
anObject(42);
```

  - If ( ) overloaded → calls overload

Other Overloads

- &&, ||, and comma operator
  - Predefined versions work for bool types
  - Recall: use "short-circuit evaluation"
  - When overloaded no longer uses short-circuit
    - Uses "complete evaluation" instead
    - Contrary to expectations
- Generally should not overload these operators

Friend Functions

- Nonmember functions
  - Recall: operator overloads as nonmembers
    - They access data through accessor and mutator functions
    - Very inefficient (overhead of calls)
- Friends can directly access private class data
  - No overhead, more efficient
- So: best to make nonmember operator overloads friends!

Friend Functions

- Friend function of a class
  - Not a member function
  - Has direct access to private members
    - Just as member functions do
- Use keyword *friend* in front of function declaration
  - Specified IN class definition
  - But they're NOT member functions!

Friend Function Uses

- Operator Overloads
  - Most common use of friends
  - Improves efficiency
  - Avoids need to call accessor/mutator member functions
  - Operator must have access anyway
    - Might as well give full access as friend
- Friends can be any function

Friend Function Purity

- Friends not pure?
  - "Spirit" of OOP dictates all operators and functions be member functions
  - Many believe friends violate basic OOP principles
- Advantageous?
  - For operators: very!
  - Allows automatic type conversion
  - Still encapsulates: friend is in class definition
  - Improves efficiency

Friend Classes

- Entire classes can be friends
  - Similar to function being friend to class
  - Example:
    class F is friend of class C
    - All class F member functions are friends of C
    - NOT reciprocated
    - Friendship granted, not taken
- Syntax: friend class F
  - Goes inside class definition of "authorizing" class

References

- Reference defined:
  - Name of a storage location
  - Similar to "pointer"
- Example of stand alone reference:
  - int robert;
    int& bob = robert;
    - *bob* is reference to storage location for *robert*
    - Changes made to *bob* will affect *robert*
- Confusing?

References Usage

- Seemingly dangerous
- Useful in several cases:
- Call-by-reference
  - Often used to implement this mechanism
- Returning a reference
  - Allows operator overload implementations to be written more naturally
  - Think of as returning an "alias" to a variable

Returning Reference

- Syntax:
  double& sampleFunction(double& variable);
  - double& and double are different
  - Must match in function declaration and heading
- Returned item must "have" a reference
  - Like a variable of that type
  - Cannot be expression like "x+5"
    - Has no place in memory to "refer to"

Returning Reference in Definition

- Example function definition:

```
double& sampleFunction(double& variable)
{
    return variable;
}
```

- Trivial, useless example
- Shows concept only
- Major use:
  - Certain overloaded operators

Overloading >> and <<

- Enables input and output of our objects
  - Similar to other operator overloads
  - New subtleties
- Improves readability
  - Like all operator overloads do
  - Enables:
    cout << myObject;
    cin >> myObject;
  - Instead of need for:
    myObject.output(); …

Overloading >>

- Insertion operator, <<
  - Used with cout
  - A binary operator
- Example:
  cout << "Hello";
  - Operator is <<
  - 1st operand is predefined object *cout*
    - From library iostream
  - 2nd operand is literal string "Hello"

Overloading >>

- Operands of >>
  - Cout object, of class type ostream
  - Our class type
- Recall Money class
  - Used member function output()
  - Nicer if we can use >> operator:

```
Money amount(100);
cout << "I have " << amount << endl;
```

instead of:

```
cout << "I have ";
amount.output()
```

Overloaded >> Return Value

- Money amount(100);
  cout << amount;
  - << should return some value
  - To allow cascades:
    cout << "I have " << amount;
    (cout << "I have ") << amount;
    - Two are equivalent
- What to return?
  - cout object!
    - Returns its first argument type, ostream

Overloaded >> Example:
Display 8.5 Overloading << and >> (1 of 5)

Display 8.5 Overloading << and >>

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

//Class for amounts of money in U.S. currency
class Money
{
public:
    Money( );
    Money(double amount);
    Money(int theDollars, int theCents);
    Money(int theDollars);
    double getAmount( ) const;
    int getDollars( ) const;
    int getCents( ) const;
    friend const Money operator +(const Money& amount1, const Money& amount2)
    friend const Money operator -(const Money& amount1, const Money& amount2)
    friend bool operator ==(const Money& amount1, const Money& amount2);
    friend const Money operator -(const Money& amount);
    friend ostream& operator <<(ostream& outputStream, const Money& amount);
    friend istream& operator >>(istream& inputStream, Money& amount);
private:
    int dollars; //A negative amount is represented as negative dollars and
    int cents; //negative cents. Negative $4.50 is represented as -4 and -50.
```

Overloaded >> Example:
Display 8.5 Overloading << and >> (2 of 5)

```
    int dollarsPart(double amount) const;
    int centsPart(double amount) const;
    int round(double number) const;
};

int main( )
{
    Money yourAmount, myAmount(10, 9);
    cout << "Enter an amount of money: ";
    cin >> yourAmount;
    cout << "Your amount is " << yourAmount << endl;
    cout << "My amount is " << myAmount << endl;

    if (yourAmount == myAmount)
        cout << "We have the same amounts.\n";
    else
        cout << "One of us is richer.\n";

    Money ourAmount = yourAmount + myAmount;
```

Overloaded >> Example:
Display 8.5 Overloading << and >> (3 of 5)

Display 8.5 Overloading << and >>

```
    cout << yourAmount << " + " << myAmount
         << " equals " << ourAmount << endl;

    Money diffAmount = yourAmount - myAmount;
    cout << yourAmount << " - " << myAmount
         << " equals " << diffAmount << endl;

    return 0;
}
```

Since << returns a reference, you can chain << like this. You can chain >> in a similar way.

<Definitions of other member functions are as in Display 8.1. Definitions of other overloaded operators are as in Display 8.3.>

```
ostream& operator <<(ostream& outputStream, const Money& amount)
{
    int absDollars = abs(amount.dollars);
    int absCents = abs(amount.cents);
    if (amount.dollars < 0 || amount.cents < 0)
        //accounts for dollars == 0 or cents == 0
        outputStream << "$-";
    else
        outputStream << '$';
    outputStream << absDollars;
```

In the `main` *function,* `cout` *is plugged in for* `outputStream`*.*

For an alternate input algorithm, see Self-Test Exercise 3 in Chapter 7.

Overloaded >> Example:
Display 8.5 Overloading << and >> (4 of 5)

```
    if (absCents >= 10)
        outputStream << '.' << absCents;
    else
        outputStream << '.' << '0' << absCents;

    return outputStream;
}

//Uses iostream and cstdlib:
istream& operator >>(istream& inputStream, Money& amount)
{
    char dollarSign;
    inputStream >> dollarSign; //hopefully
    if (dollarSign != '$')
    {
        cout << "No dollar sign in Money input.\n";
        exit(1);
    }

    double amountAsDouble;
    inputStream >> amountAsDouble;
    amount.dollars = amount.dollarsPart(amountAsDouble);
```

Returns a reference

In the **main** *function,* **cin** *is plugged in for* **inputStream**.

Since this is not a member operator, you need to specify a calling object for member functions of **Money**.

(continued)

Overloaded >> Example:
Display 8.5 Overloading << and >> (5 of 5)

Display 8.5 Overloading << and >>

```
    amount.cents = amount.centsPart(amountAsDouble);

    return inputStream;
}
```

Returns a reference

SAMPLE DIALOGUE

Enter an amount of money: **$123.45**
Your amount is $123.45
My amount is $10.09.
One of us is richer.
$123.45 + $10.09 equals $133.54
$123.45 - $10.09 equals $113.36

Assignment Operator, =

- Must be overloaded as member operator
- Automatically overloaded
  - Default assignment operator:
    - Member-wise copy
    - Member variables from one object → corresponding member variables from other
- Default OK for simple classes
  - But with pointers → must write our own!

Increment and Decrement

- Each operator has two versions
  - Prefix notation: ++x;
  - Postfix notation: x++;
- Must distinguish in overload
  - Standard overload method → Prefix
  - Add 2d parameter of type int → Postfix
    - Just a marker for compiler!
    - Specifies postfix is allowed

Overload Array Operator, [ ]

- Can overload [ ] for your class
  - To be used with objects of your class
  - Operator must return a reference!
  - Operator [ ] must be a member function!

Summary 1

- C++ built-in operators can be overloaded
  - To work with objects of your class
- Operators are really just functions
- Friend functions have direct private member access
- Operators can be overloaded as member functions
  - 1st operand is calling object

Summary 2

- Friend functions add efficiency only
  - Not required if sufficient accessors/mutators available
- Reference "names" a variable with an alias
- Can overload <<, >>
  - Return type is a reference to stream type

Learning Objectives

- An Array Type for Strings
  - C-Strings
- Character Manipulation Tools
  - Character I/O
  - get, put member functions
  - putback, peek, ignore
- Standard Class string
  - String processing

Introduction

- Two string types:
- C-strings
  - Array with base type char
  - End of string marked with null, "\0"
  - "Older" method inherited from C
- String class
  - Uses templates

C-Strings

- Array with base type *char*
  - One character per indexed variable
  - One extra character: "\0"
    - Called "null character"
    - End marker
- We've used c-strings
  - Literal "Hello" stored as c-string

C-String Variable

- Array of characters:
  char s[10];
  - Declares a c-string variable to hold up to 9 characters
  - + one null character
- Typically "partially-filled" array
  - Declare large enough to hold max-size string
  - Indicate end with null
- Only difference from standard array:
  - Must contain null character

C-String Storage

- A standard array:
  char s[10];
  - If s contains string "Hi Mom", stored as:

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]
H	i		M	o	m	!	\0	?	?

C-String Initialization

- Can initialize c-string:
  char myMessage[20] = "Hi there.";
  - Needn’t fill entire array
  - Initialization places "\0" at end
- Can omit array-size:
  char shortString[] = "abc";
  - Automatically makes size one more than length of quoted string
  - NOT same as:
    char shortString[] = {"a", "b", "c"};

C-String Indexes

- A c-string IS an array
- Can access indexed variables of:
  char ourString[5] = "Hi";
  - ourString[0] is "H"
  - ourString[1] is "i"
  - ourString[2] is "\0"
  - ourString[3] is unknown
  - ourString[4] is unknown

C-String Index Manipulation

- Can manipulate indexed variables
  char happyString[7] = "DoBeDo";
  happyString[6] = "Z";
  - Be careful!
  - Here, "\0" (null) was overwritten by a "Z"!
- If null overwritten, c-string no longer "acts" like c-string!
  - Unpredictable results!

Library

- Declaring c-strings
  - Requires no C++ library
  - Built into standard C++
- Manipulations
  - Require library <cstring>
  - Typically included when using c-strings
    - Normally want to do "fun" things with them

= and == with C-strings

- C-strings not like other variables
  - Cannot assign or compare:
    char aString[10];
    aString = "Hello"; // ILLEGAL!
    - Can ONLY use "=" at declaration of c-string!
- Must use library function for assignment:
  strcpy(aString, "Hello");
  - Built-in function (in <cstring>)
  - Sets value of aString equal to "Hello"
  - NO checks for size!
    - Up to programmer, just like other arrays!

Comparing C-strings

- Also cannot use operator ==

```
char aString[10] = "Hello";
char anotherString[10] = "Goodbye";
```

  – aString == anotherString; // NOT allowed!

- Must use library function again:

```
if (strcmp(aString, anotherString))
    cout << "Strings NOT same.";
else
    cout << "Strings are same.";
```

The <cstring> Library: **Display 9.1** Some Predefined C-String Functions in <cstring> (1 of 2)

- Full of string manipulation functions

Display 9.1 Some Predefined C-String Functions in `<cstring>`

FUNCTION	DESCRIPTION	CAUTIONS
`strcpy(`*Target_String_Var*, *Src_String*`)`	Copies the C-string value *Src_String* into the C-string variable *Target_String_Var*.	Does not check to make sure *Target_String_Var* is large enough to hold the value *Src_String*.
`strcpy(`*Target_String_Var*, *Src_String*, *Limit*`)`	The same as the two-argument `strcpy` except that at most *Limit* characters are copied.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcpy`. Not implemented in all versions of C++.
`strcat(`*Target_String_Var*, *Src_String*`)`	Concatenates the C-string value *Src_String* onto the end of the C-string in the C-string variable *Target_String_Var*.	Does not check to see that *Target_String_Var* is large enough to hold the result of the concatenation.

(continued)

The <cstring> Library: **Display 9.1** Some Predefined C-String Functions in <cstring> (2 of 2)

Display 9.1 Some Predefined C-String Functions in `<cstring>`

FUNCTION	DESCRIPTION	CAUTIONS
`strcat(`*Target_String_Var*, *Src_String*, *Limit*`)`	The same as the two argument `strcat` except that at most *Limit* characters are appended.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcat`. Not implemented in all versions of C++.
`strlen(`*Src_String*`)`	Returns an integer equal to the length of *Src_String*. (The null character, `'\0'`, is not counted in the length.)	
`strcmp(`*String_1*, *String_2*`)`	Returns 0 if *String_1* and *String_2* are the same. Returns a value < 0 if *String_1* is less than *String_2*. Returns a value > 0 if *String_1* is greater than *String_2* (that is, returns a nonzero value if *String_1* and *String_2* are different). The order is lexicographic.	If *String_1* equals *String_2*, this function returns 0, which converts to `false`. Note that this is the reverse of what you might expect it to return when the strings are equal.
`strcmp(`*String_1*, *String_2*, *Limit*`)`	The same as the two-argument `strcat` except that at most *Limit* characters are compared.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcmp`. Not implemented in all versions of C++.

C-string Functions: strlen()

- "String length"
- Often useful to know string length:

```
char myString[10] = "dobedo";
cout << strlen(myString);
```

  - Returns number of characters
    - Not including null
  - Result here:

    6

C-string Functions: strcat()

- strcat()
- "String concatenate":
  char stringVar[20] = "The rain";
  strcat(stringVar, "in Spain");
  - Note result:
    stringVar now contains "The rainin Spain"
  - Be careful!
  - Incorporate spaces as needed!

C-string Arguments and Parameters

- Recall: c-string is array
- So c-string parameter is array parameter
  - C-strings passed to functions can be changed by receiving function!
- Like all arrays, typical to send size as well
  - Function "could" also use "\0" to find end
  - So size not necessary if function won't change c-string parameter
  - Use "const" modifier to protect c-string arguments

C-String Output

- Can output with insertion operator, <<
- As we've been doing already:
  cout << news << " Wow.\n";
  - Where *news* is a c-string variable
- Possible because << operator is overloaded for c-strings!

C-String Input

- Can input with extraction operator, >>
  - Issues exist, however
- Whitespace is "delimiter"
  - Tab, space, line breaks are "skipped"
  - Input reading "stops" at delimiter
- Watch size of c-string
    - Must be large enough to hold entered string!
    - C++ gives no warnings of such issues!

C-String Input Example

- char a[80], b[80];
  cout << "Enter input: ";
  cin >> a >> b;
  cout << a << b << "END OF OUTPUT\n";
- Dialogue offered:

  Enter input: <u>Do be do to you!</u>
  DobeEND OF OUTPUT

  – Note: Underlined portion typed at keyboard
- C-string *a* receives: "do"
- C-string *b* receives: "be"

C-String Line Input

- Can receive entire line into c-string
- Use getline(), a predefined member function:

```
char a[80];
cout << "Enter input: ";
cin.getline(a, 80);
cout << a << "END OF OUTPUT\n";
```

  – Dialogue:
    Enter input: <u>Do be do to you!</u>
    Do be do to you!END OF INPUT

Example: Command Line Arguments

- Programs invoked from the command line (e.g. a UNIX shell, DOS command prompt) can be sent arguments
  - Example: COPY C:\FOO.TXT D:\FOO2.TXT
    - This runs the program named “COPY” and sends in two C-String parameters, “C:\FOO.TXT” and “D:\FOO2.TXT”
    - It is up to the COPY program to process the inputs presented to it; i.e. actually copy the files
- Arguments are passed as an array of C-Strings to the main function

Example: Command Line Arguments

- Header for main
  - int main(int argc, char *argv[])
  - argc specifies how many arguments are supplied. The name of the program counts, so argc will be at least 1.
  - argv is an array of C-Strings.
    - argv[0] holds the name of the program that is invoked
    - argv[1] holds the name of the first parameter
    - argv[2] holds the name of the second parameter
    - Etc.

Example: Command Line Arguments

```
// Echo back the input arguments
int main(int argc, char *argv[])
{
  for (int i=0; i<argc; i++)
  {
    cout << "Argument " << i << " " << argv[i] << endl;
  }
  return 0;
}
```

Sample Execution

```
> Test
Argument 0 Test
```

Invoking Test from command prompt

Sample Execution

```
> Test hello world
Argument 0 Test
Argument 1 hello
Argument 2 world
```

More getline()

- Can explicitly tell length to receive:

```
char shortString[5];
cout << "Enter input: ";
cin.getline(shortString, 5);
cout << shortString << "END OF OUTPUT\n";
```

  - Results:

    Enter input: dobedowap
    dobeEND OF OUTPUT

  - Forces FOUR characters only be read
    - Recall need for null character!

Character I/O

- Input and output data
  - ALL treated as character data
  - e.g., number 10 outputted as "1" and "0"
  - Conversion done automatically
    - Uses low-level utilities
- Can use same low-level utilities ourselves as well

Member Function get()

- Reads one char at a time
- Member function of cin object:
  char nextSymbol;
  cin.get(nextSymbol);
  - Reads next char & puts in variable nextSymbol
  - Argument must be char type
    - Not "string"!

Member Function put()

- Outputs one character at a time
- Member function of cout object:
- Examples:

```
cout.put("a");
```

  – Outputs letter "a" to screen

```
char myString[10] = "Hello";
cout.put(myString[1]);
```

  – Outputs letter "e" to screen

More Member Functions

- putback()
  - Once read, might need to "put back"
  - cin.putback(lastChar);
- peek()
  - Returns next char, but leaves it there
  - peekChar = cin.peek();
- ignore()
  - Skip input, up to designated character
  - cin.ignore(1000, "\n");
    - Skips at most 1000 characters until "\n"

Character-Manipulating Functions: **Display 9.3** Some Functions in <cctype> (1 of 3)

Display 9.3 Some Functions in <cctype>

FUNCTION	DESCRIPTION	EXAMPLE

`toupper(`*Char_Exp*`)`	Returns the uppercase version of *Char_Exp* (as a value of type `int`).	`char c = toupper('a');`
		`cout << c;`
		Outputs: A

`tolower(`*Char_Exp*`)`	Returns the lowercase version of *Char_Exp* (as a value of type `int`).	`char c = tolower('A');`
		`cout << c;`
		Outputs: a

`isupper(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is an uppercase letter; otherwise, returns `false`.	`if (isupper(c))`
		`  cout << "Is uppercase.";`
		`else`
		`  cout << "Is not uppercase.";`

Character-Manipulating Functions: **Display 9.3** Some Functions in <cctype> (2 of 3)

Display 9.3 Some Functions in <cctype>

FUNCTION	DESCRIPTION	EXAMPLE

`islower(Char_Exp)`	Returns `true` provided *Char_Exp* is a lowercase letter; otherwise, returns `false`.	`char c = 'a';`
		`if (islower(c))`
		`  cout << c << " is lowercase.";`
		Outputs: `a is lowercase.`

`isalpha(Char_Exp)`	Returns `true` provided *Char_Exp* is a letter of the alphabet; otherwise, returns `false`.	`char c = '$';`
		`if (isalpha(c))`
		`  cout << "Is a letter.";`
		`else`
		`  cout << "Is not a letter.";`
		Outputs: `Is not a letter.`

`isdigit(Char_Exp)`	Returns `true` provided *Char_Exp* is one of the digits `'0'` through `'9'`; otherwise, returns `false`.	`if (isdigit('3'))`
		`  cout << "It's a digit.";`
		`else`
		`  cout << "It's not a digit.";`
		Outputs: `It's a digit.`

`isalnum(Char_Exp)`	Returns `true` provided *Char_Exp* is either a letter or a digit; otherwise, returns `false`.	`if (isalnum('3') && isalnum('a'))`
		`  cout << "Both alphanumeric.";`
		`else`
		`  cout << "One or more are not.";`
		Outputs: `Both alphanumeric.`

Character-Manipulating Functions: **Display 9.3** Some Functions in <cctype> (3 of 3)

`isspace(Char_Exp)`	Returns `true` provided *Char_Exp* is a whitespace character, such as the blank or newline character; otherwise, returns `false`.	`//Skips over one "word" and sets c`
		`//equal to the first whitespace`
		`//character after the "word":`
		`do`
		`{`
		`  cin.get(c);`
		`} while (! isspace(c));`

`ispunct(Char_Exp)`	Returns `true` provided *Char_Exp* is a printing character other than whitespace, a digit, or a letter; otherwise, returns `false`.	`if (ispunct('?'))`
		`  cout << "Is punctuation.";`
		`else`
		`  cout << "Not punctuation.";`

`isprint(Char_Exp)`	Returns `true` provided *Char_Exp* is a printing character; otherwise, returns `false`.	
`isgraph(Char_Exp)`	Returns `true` provided *Char_Exp* is a printing character other than whitespace; otherwise, returns `false`.	
`isctrl(Char_Exp)`	Returns `true` provided *Char_Exp* is a control character; otherwise, returns `false`.	

Standard Class string

- Defined in library:
  #include <string>
  using namespace std;
- String variables and expressions
  - Treated much like simple types
- Can assign, compare, add:
  string s1, s2, s3;
  s3 = s1 + s2; //Concatenation
  s3 = "Hello Mom!" //Assignment
  - Note c-string "Hello Mom!" automatically converted to string type!

Display 9.4
Program Using the Class string

Display 9.4 Program Using the Class string

```
//Demonstrates the standard class string.
#include <iostream>
#include <string>
using namespace std;

int main( )
{
    string phrase;
    string adjective("fried"), noun("ants");
    string wish = "Bon appetite!";

    phrase = "I love " + adjective + " " + noun + "!";
    cout << phrase << endl
         << wish << endl;

    return 0;
}
```

Initialized to the empty string.

Two equivalent ways of initializing a string variable

SAMPLE DIALOGUE

```
I love fried ants!
Bon appetite!
```

I/O with Class string

- Just like other types!
- string s1, s2;
  cin >> s1;
  cin >> s2;
- Results:
  User types in:
  May the hair on your toes grow long and curly!
- Extraction still ignores whitespace:
  s1 receives value "May"
  s2 receives value "the"

getline() with Class string

- For complete lines:

```
string line;
cout << "Enter a line of input: ";
getline(cin, line);
cout << line << "END OF OUTPUT";
```

- Dialogue produced:

Enter a line of input: <u>Do be do to you!</u>
Do be do to you!END OF INPUT

  – Similar to c-string's usage of getline()

Other getline() Versions

- Can specify "delimiter" character:
  ```
  string line;
  cout << "Enter input: ";
  getline(cin, line, "?");
  ```
  – Receives input until "?" encountered
- getline() actually returns reference
  – string s1, s2;
    getline(cin, s1) >> s2;
  – Results in: (cin) >> s2;

Pitfall: Mixing Input Methods

- Be careful mixing cin >> var and getline
  - int n;
    string line;
    cin >> n;
    getline(cin, line);
  - If input is: 42
    Hello hitchhiker.
    - Variable n set to 42
    - line set to empty string!
  - cin >> n skipped leading whitespace, leaving "\n" on stream for getline()!

Class string Processing

- Same operations available as c-strings
- And more!
  - Over 100 members of standard string class
- Some member functions:
  - .length()
    - Returns length of string variable
  - .at(i)
    - Returns reference to char at position i

Display 9.7 Member Functions of the Standard Class string (1 of 2)

Display 9.7 Member Functions of the Standard Class `string`

EXAMPLE	REMARKS
Constructors	
`string str;`	Default constructor; creates empty `string` object `str`.
`string str("string");`	Creates a `string` object with data `"string"`.
`string str(aString);`	Creates a `string` object `str` that is a copy of `aString`. `aString` is an object of the class `string`.
Element access	
`str[i]`	Returns read/write reference to character in `str` at index `i`.
`str.at(i)`	Returns read/write reference to character in `str` at index `i`.
`str.substr(position, length)`	Returns the substring of the calling object starting at `position` and having `length` characters.
Assignment/Modifiers	
`str1 = str2;`	Allocates space and initializes it to `str2`'s data, releases memory allocated for `str1`, and sets `str1`'s size to that of `str2`.
`str1 += str2;`	Character data of `str2` is concatenated to the end of `str1`; the size is set appropriately.
`str.empty( )`	Returns `true` if `str` is an empty `string`; returns `false` otherwise.

(continued)

Display 9.7 Member Functions of the Standard Class string (2 of 2)

Display 9.7 Member Functions of the Standard Class `string`

EXAMPLE	REMARKS
`str1 + str2`	Returns a `string` that has `str2`'s data concatenated to the end of `str1`'s data. The size is set appropriately.
`str.insert(pos, str2)`	Inserts `str2` into `str` beginning at position `pos`.
`str.remove(pos, length)`	Removes substring of size `length`, starting at position `pos`.
Comparisons	
`str1 == str2  str1 != str2`	Compare for equality or inequality; returns a Boolean value.
`str1 < str2  str1 > str2`	Four comparisons. All are lexicographical comparisons.
`str1 <= str2  str1 >= str2`	
`str.find(str1)`	Returns index of the first occurrence of `str1` in `str`.
`str.find(str1, pos)`	Returns index of the first occurrence of string `str1` in `str`; the search starts at position `pos`.
`str.find_first_of(str1, pos)`	Returns the index of the first instance in `str` of any character in `str1`, starting the search at position `pos`.
`str.find_first_not_of (str1, pos)`	Returns the index of the first instance in `str` of any character *not* in `str1`, starting search at position `pos`.

C-string and string Object Conversions

- Automatic type conversions
  - From c-string to string object:
    char aCString[] = "My C-string";
    string stringVar;
    stringVar = aCstring;
    - Perfectly legal and appropriate!
  - aCString = stringVar;
    - ILLEGAL!
    - Cannot auto-convert to c-string
  - Must use explicit conversion:
    strcpy(aCString, stringVar.c_str());

Converting between `string` and numbers

- In C++11 it is simply a matter of calling **stof**, **stod**, **stoi**, or **stol** to convert a string to a float, double, int, or long, respectively.

```
int i;
double d;
string s;
i = stoi("35");  // Converts the string "35" to an integer 35
d = stod("2.5"); // Converts the string "2.5" to the double 2.5
```

Converting between numbers and `string` objects

- In C++11 use **to_string** to convert a numeric type to a string

```
 string s;
 s = to_string(d*2);  // Converts the
double 5.0 to a
                      // string "5.0000"
```

Summary

- C-string variable is "array of characters"
  - With addition of null character, "\0"
- C-strings act like arrays
  - Cannot assign, compare like simple variables
- Libraries <cctype> & <string> have useful manipulating functions
- cin.get() reads next single character
- getline() versions allow full line reading
- Class string objects are better-behaved than c-strings

Learning Objectives

- Pointers
  - Pointer variables
  - Memory management
- Dynamic Arrays
  - Creating and using
  - Pointer arithmetic
- Classes, Pointers, Dynamic Arrays
  - The *this* pointer
  - Destructors, copy constructors

Pointer Introduction

- Pointer definition:
  - Memory address of a variable
- Recall: memory divided
  - Numbered memory locations
  - Addresses used as name for variable
- You've used pointers already!
  - Call-by-reference parameters
    - Address of actual argument was passed

Pointer Variables

- Pointers are "typed"
  - Can store pointer in variable
  - Not int, double, etc.
    - Instead: A POINTER to int, double, etc.!
- Example:
  double *p;
  - p is declared a "pointer to double" variable
  - Can hold pointers to variables of type double
    - Not other types! (unless typecast, but could be dangerous)

Declaring Pointer Variables

- Pointers declared like other types
  - Add "*" before variable name
  - Produces "pointer to" that type
- "*" must be before each variable
- int *p1, *p2, v1, v2;
  - p1, p2 hold pointers to int variables
  - v1, v2 are ordinary int variables

Addresses and Numbers

- Pointer is an address
- Address is an integer
- Pointer is NOT an integer!
  - Not crazy → abstraction!
- C++ forces pointers be used as addresses
  - Cannot be used as numbers
  - Even though it "is a" number

Pointing

- Terminology, view
  - Talk of "pointing", not "addresses"
  - Pointer variable "points to" ordinary variable
  - Leave "address" talk out
- Makes visualization clearer
  - "See" memory references
    - Arrows

Pointing to …

- int *p1, *p2, v1, v2;
  p1 = &v1;
  - Sets pointer variable p1 to "point to" int variable v1
- Operator, &
  - Determines "address of" variable
- Read like:
  - "p1 equals address of v1"
  - Or "p1 points to v1"

Pointing to …

- Recall:
  int *p1, *p2, v1, v2;
  p1 = &v1;
- Two ways to refer to v1 now:
  - Variable v1 itself:
    cout << v1;
  - Via pointer p1:
    cout *p1;
- Dereference operator, *
  - Pointer variable "derereferenced"
  - Means: "Get data that p1 points to"

"Pointing to" Example

- Consider:
```
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```
- Produces output:
```
42
42
```
- p1 and v1 refer to same variable

& Operator

- The "address of" operator
- Also used to specify call-by-reference parameter
  - No coincidence!
  - Recall: call-by-reference parameters pass
    "address of" the actual argument
- Operator’s two uses are closely related

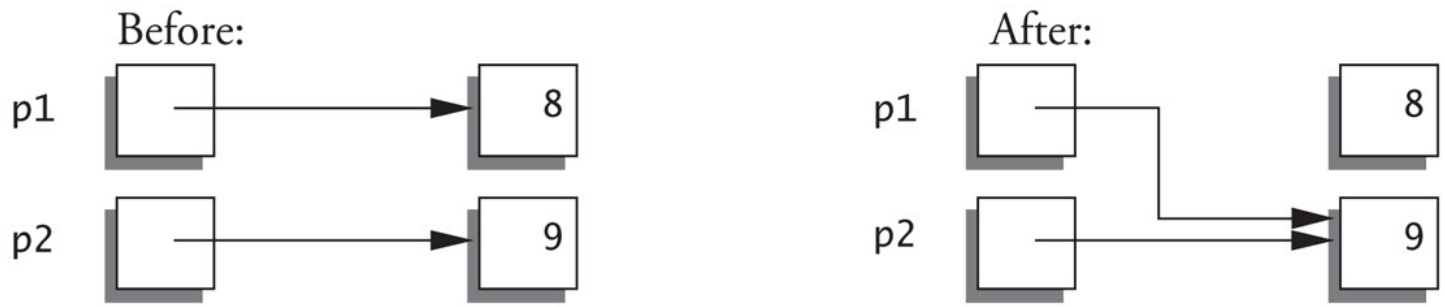
Pointer Assignments

- Pointer variables can be "assigned":
  int *p1, *p2;
  p2 = p1;
  - Assigns one pointer to another
  - "Make p2 point to where p1 points"
- Do not confuse with:
  *p1 = *p2;
  - Assigns "value pointed to" by p1, to "value
    pointed to" by p2

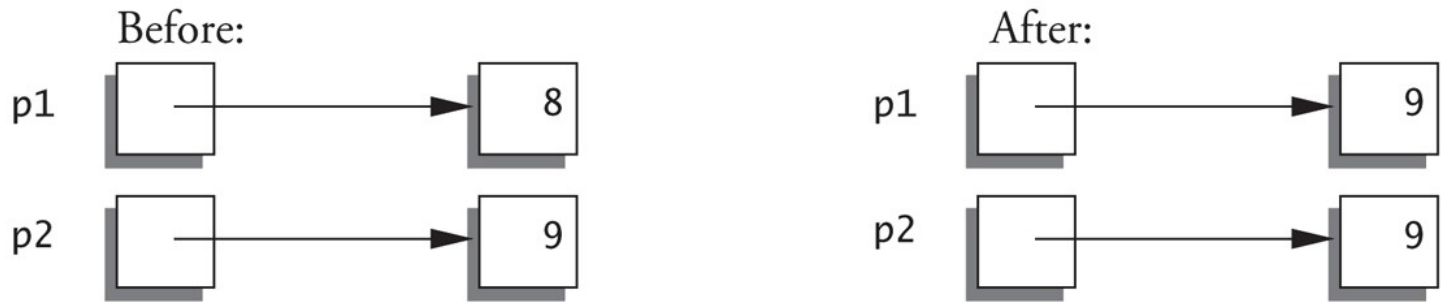
Pointer Assignments Graphic: **Display 10.1** Uses of the Assignment Operator with Pointer Variables

Display 10.1 Uses of the Assignment Operator with Pointer Variables

```
p1 = p2;
```




```
*p1 = *p2;
```



The new Operator

- Since pointers can refer to variables…
  - No "real" need to have a standard identifier
- Can dynamically allocate variables
  - Operator *new* creates variables
    - No identifiers to refer to them
    - Just a pointer!
- p1 = new int;
  - Creates new "nameless" variable, and assigns p1 to "point to" it
  - Can access with *p1
    - Use just like ordinary variable

Basic Pointer Manipulations Example: **Display 10.2** Basic Pointer Manipulations (1 of 2)

Display 10.2 Basic Pointer Manipulations

```
//Program to demonstrate pointers and dynamic variables.
#include <iostream>
using std::cout;
using std::endl;

int main( )
{
    int *p1, *p2;

    p1 = new int;
    *p1 = 42;
    p2 = p1;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    *p2 = 53;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;
```

Basic Pointer Manipulations Example: **Display 10.2** Basic Pointer Manipulations (2 of 2)

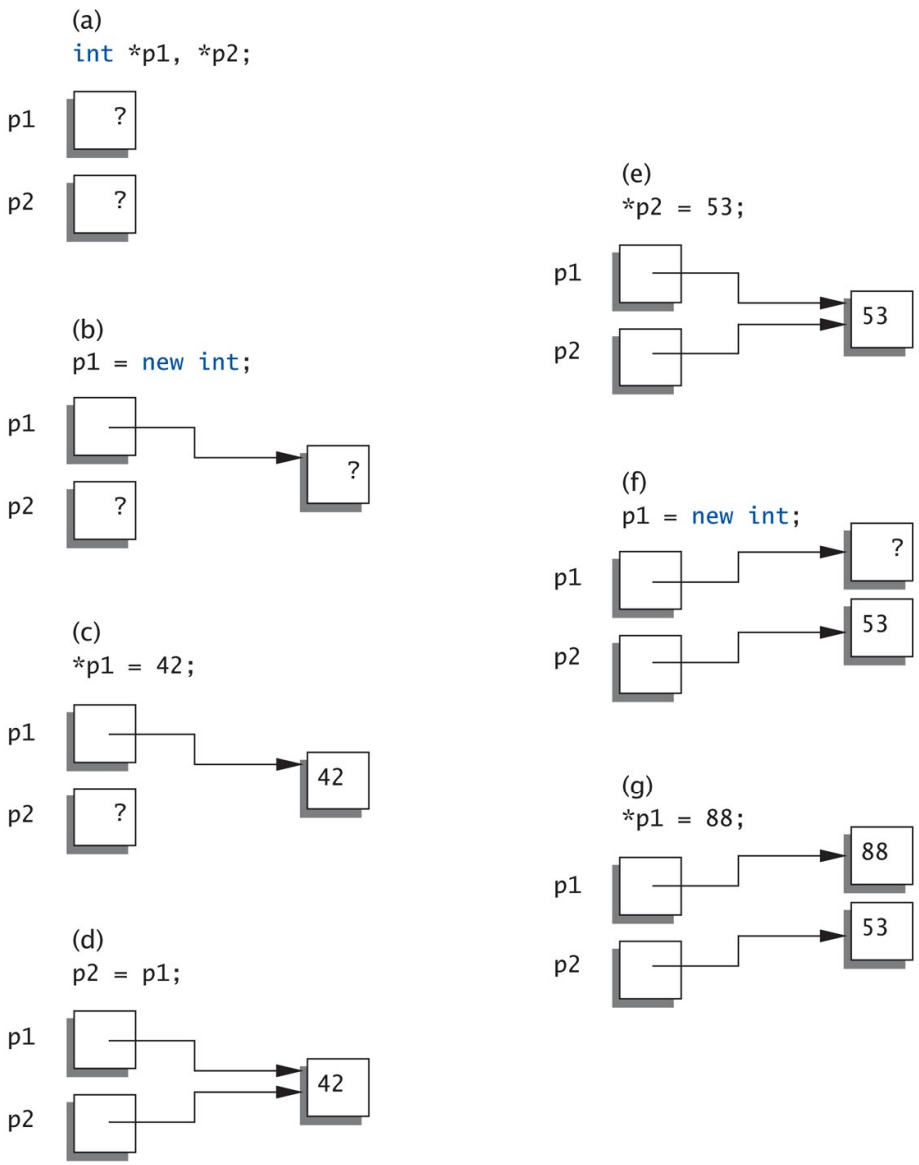
```
    p1 = new int;
    *p1 = 88;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;


    cout << "Hope you got the point of this example!\n";
    return 0;
}
```

SAMPLE DIALOGUE

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
```

Basic Pointer Manipulations Graphic: **Display 10.3** Explanation of Display 10.2

Display 10.3 Explanation of Display 10.2



More on new Operator

- Creates new dynamic variable
- Returns pointer to the new variable
- If type is class type:
  - Constructor is called for new object
  - Can invoke different constructor with initializer arguments:

```
MyClass *mcPtr;
mcPtr = new MyClass(32.0, 17);
```

- Can still initialize non-class types:

```
int *n;
n = new int(17);    //Initializes *n to 17
```

Pointers and Functions

- Pointers are full-fledged types
  - Can be used just like other types
- Can be function parameters
- Can be returned from functions
- Example:
  int* findOtherPointer(int* p);
  - This function declaration:
    - Has "pointer to an int" parameter
    - Returns "pointer to an int" variable

Memory Management

- Heap
  - Also called "freestore"
  - Reserved for dynamically-allocated variables
  - All new dynamic variables consume memory in freestore
    - If too many → could use all freestore memory
- Future "new" operations will fail if freestore is "full"

Checking new Success

- Older compilers:
  - Test if null returned by call to *new:*

```
int *p;
p = new int;
if (p == NULL)   // NULL represents empty pointer
{
    cout << "Error: Insufficient memory.\n";
    exit(1);
}
```

  - If new succeeded, program continues

new Success – New Compiler

- Newer compilers:
  - If new operation fails:
    - Program terminates automatically
    - Produces error message
- Still good practice to use NULL check
- NULL represents the empty pointer or a pointer to nothing and will be used later to mark the end of a list

C++11 nullptr

- NULL is actually the number 0 and can lead to ambiguity

```
void func(int *p);
void func(int i);
```

- Which func is invoked given **func(NULL)**? Both are equally valid since NULL is 0
- C++11 resolves this problem by introducing a new constant, **nullptr**
- **nullptr** is not 0
- Can use anywhere you could use NULL

Freestore Size

- Varies with implementations
- Typically large
  - Most programs won’t use all memory
- Memory management
  - Still good practice
  - Solid software engineering principle
  - Memory IS finite
    - Regardless of how much there is!

delete Operator

- De-allocate dynamic memory
  - When no longer needed
  - Returns memory to freestore
  - Example:

```
int *p;
p = new int(5);
… //Some processing…
delete p;
```

  - De-allocates dynamic memory "pointed to by pointer p"
    - Literally "destroys" memory

Dangling Pointers

- delete p;
  - Destroys dynamic memory
  - But p still points there!
    - Called "dangling pointer"
  - If p is then dereferenced (*p)
    - Unpredicatable results!
    - Often disastrous!
- Avoid dangling pointers
  - Assign pointer to NULL after delete:

```
delete p;
p = NULL;
```

Dynamic and Automatic Variables

- Dynamic variables
  - Created with new operator
  - Created and destroyed while program runs
- Local variables
  - Declared within function definition
  - Not dynamic
    - Created when function is called
    - Destroyed when function call completes
  - Often called "automatic" variables
    - Properties controlled for you

Define Pointer Types

- Can "name" pointer types
- To be able to declare pointers like other variables
  - Eliminate need for "*" in pointer declaration
- typedef int* IntPtr;
  - Defines a "new type" alias
  - Consider these declarations:
    IntPtr p;
    int *p;
    - The two are equivalent

Pitfall: Call-by-value Pointers

- Behavior subtle and troublesome
  - If function changes pointer parameter itself → only change is to local copy
- Best illustrated with example…

Call-by-value Pointers Example: **Display 10.4** A Call-by-Value Pointer Parameter (1 of 2)

Display 10.4 A Call-by-Value Pointer Parameter

```
//Program to demonstrate the way call-by-value parameters
//behave with pointer arguments.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

typedef int* IntPointer;

void sneaky(IntPointer temp);

int main( )
{
    IntPointer p;

    p = new int;
    *p = 77;
    cout << "Before call to function *p == "
         << *p << endl;
```

Call-by-value Pointers Example: **Display 10.4** A Call-by-Value Pointer Parameter (2 of 2)

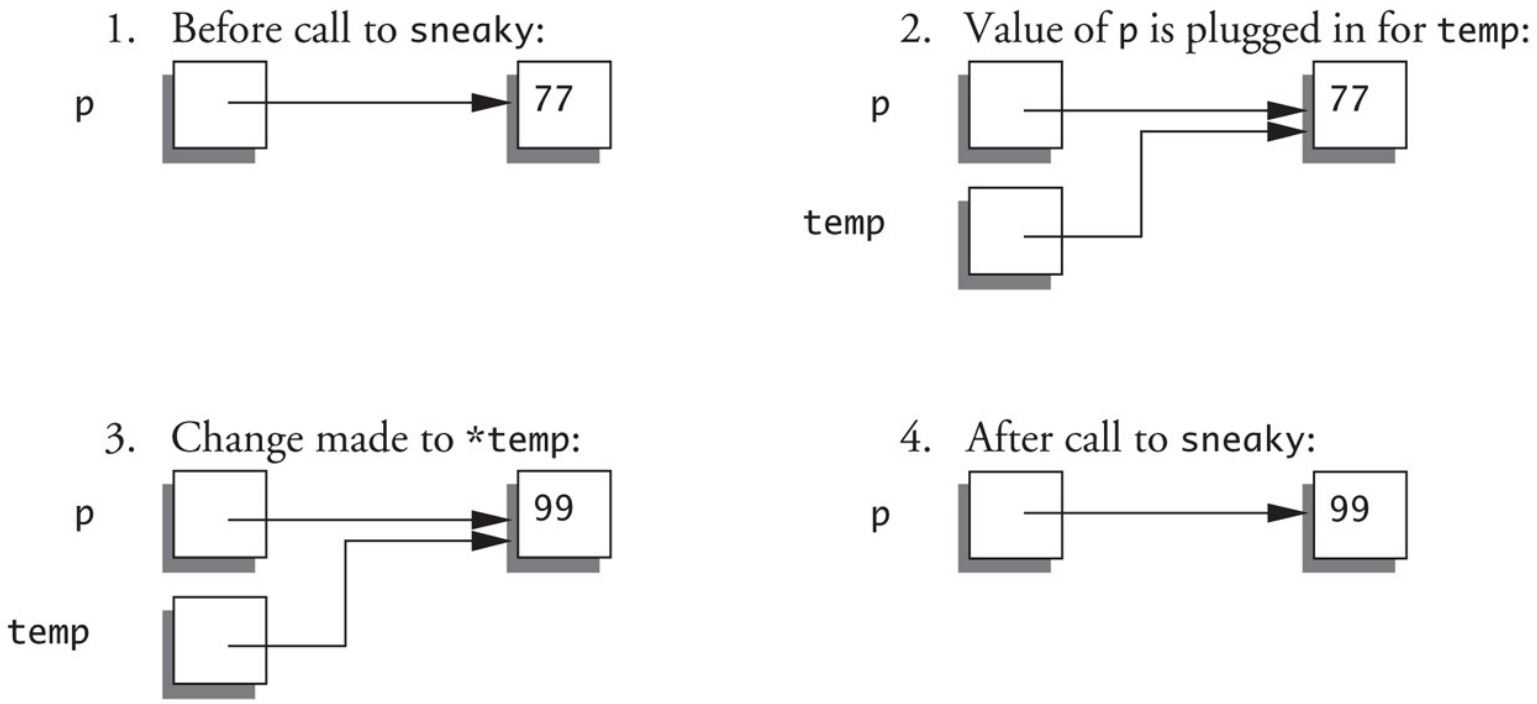
```
    sneaky(p);

    cout << "After call to function *p == "
         << *p << endl;

    return 0;
}
void sneaky(IntPointer temp)
{
    *temp = 99;
    cout << "Inside function call *temp == "
         << *temp << endl;
}
```

SAMPLE DIALOGUE

```
Before call to function *p == 77
Inside function call *temp == 99
After call to function *p == 99
```

Call-by-value Pointers Graphic: **Display 10.5** The Function Call sneaky(p);

Display 10.5 The Function Call sneaky(p);



Dynamic Arrays

- Array variables
  - Really pointer variables!
- Standard array
  - Fixed size
- Dynamic array
  - Size not specified at programming time
  - Determined while program running

Array Variables

- Recall: arrays stored in memory addresses, sequentially
  - Array variable "refers to" first indexed variable
  - So array variable is a kind of pointer variable!
- Example:
  int a[10];
  int * p;
  - a and p are both pointer variables!

Array Variables → Pointers

- Recall previous example:
  ```
  int a[10];
  typedef int* IntPtr;
  IntPtr p;
  ```
- a and p are pointer variables
  - Can perform assignments:
    p = a; // Legal.
    - p now points where a points
      - To first indexed variable of array a
  - a = p; // ILLEGAL!
    - Array pointer is CONSTANT pointer!

Array Variables → Pointers

- Array variable
  int a[10];
- MORE than a pointer variable
  - "const int *" type
  - Array was allocated in memory already
  - Variable *a* MUST point there…always!
    - Cannot be changed!
- In contrast to ordinary pointers
  - Which can (& typically do) change

Dynamic Arrays

- Array limitations
  - Must specify size first
  - May not know until program runs!
- Must "estimate" maximum size needed
  - Sometimes OK, sometimes not
  - "Wastes" memory
- Dynamic arrays
  - Can grow and shrink as needed

Creating Dynamic Arrays

- Very simple!
- Use new operator
  - Dynamically allocate with pointer variable
  - Treat like standard arrays
- Example:

```
typedef double * DoublePtr;
DoublePtr d;
d = new double[10];    //Size in brackets
```

  - Creates dynamically allocated array variable *d*, with ten elements, base type double

Deleting Dynamic Arrays

- Allocated dynamically at run-time
  - So should be destroyed at run-time
- Simple again. Recall Example:

```
d = new double[10];
… //Processing
delete [] d;
```

  - De-allocates all memory for dynamic array
  - Brackets indicate "array" is there
  - Recall: *d* still points there!
    - Should set d = NULL;

Function that Returns an Array

- Array type NOT allowed as return-type of function
- Example:
  int [] someFunction(); // ILLEGAL!
- Instead return pointer to array base type:
  int* someFunction(); // LEGAL!

Pointer Arithmetic

- Can perform arithmetic on pointers
  - "Address" arithmetic
- Example:
  typedef double* DoublePtr;
  DoublePtr d;
  d = new double[10];
  - d contains address of d[0]
  - d + 1 evaluates to address of d[1]
  - d + 2 evaluates to address of d[2]
    - Equates to "address" at these locations

Alternative Array Manipulation

- Use pointer arithmetic!
- "Step thru" array without indexing:

```
for (int i = 0; i < arraySize; i++)
      cout << *(d + I) << " " ;
```

- Equivalent to:

```
for (int i = 0; i < arraySize; i++)
      cout << d[I] << " " ;
```

- Only addition/subtraction on pointers
  - No multiplication, division
- Can use ++ and -- on pointers

Multidimensional Dynamic Arrays

- Yes we can!
- Recall: "arrays of arrays"
- Type definitions help "see it":
  typedef int* IntArrayPtr;
  IntArrayPtr *m = new IntArrayPtr[3];
  - Creates array of three pointers
  - Make each allocate array of 4 ints
- for (int i = 0; i < 3; i++)
  m[i] = new int[4];
  - Results in three-by-four dynamic array!

Back to Classes

- The -> operator
  - Shorthand notation
- Combines dereference operator, *, and dot operator
- Specifies member of class "pointed to" by given pointer
- Example:

```
MyClass *p;
p = new MyClass;
p->grade = "A";   Equivalent to:
(*p).grade = "A";
```

The this Pointer

- Member function definitions might need to refer to calling object
- Use predefined *this* pointer
  - Automatically points to calling object:

```
Class Simple
{
public:
  void showStuff() const;
private:
  int stuff;
};
```

- Two ways for member functions to access:

```
cout << stuff;
cout << this->stuff;
```

Overloading Assignment Operator

- Assignment operator returns reference
  - So assignment "chains" are possible
  - e.g., a = b = c;
    - Sets a and b equal to c
- Operator must return "same type" as it's left-hand side
  - To allow chains to work
  - The *this* pointer will help with this!

Overloading Assignment Operator

- Recall: Assignment operator must be member of the class
  - It has one parameter
  - Left-operand is calling object
    s1 = s2;
    - Think of like: s1.=(s2);
- s1 = s2 = s3;
  - Requires (s1 = s2) = s3;
  - So (s1 = s2) must return object of s1"s type
    - And pass to " = s3";

Overloaded = Operator Definition

- Uses string Class example:

```
StringClass& StringClass::operator=(const StringClass& rtSide)
{
	if (this == &rtSide)		// if right side same as left side
		return *this;
	else
	{
		capacity = rtSide.length;
		length
		length = rtSide.length;
		delete [] a;
		a = new char[capacity];
		for (int I = 0; I < length; I++)
			a[I] = rtSide.a[I];
		return *this;
	}
}
```

Shallow and Deep Copies

- Shallow copy
  - Assignment copies only member variable contents over
  - Default assignment and copy constructors
- Deep copy
  - Pointers, dynamic memory involved
  - Must dereference pointer variables to "get to" data for copying
  - Write your own assignment overload and copy constructor in this case!

Destructor Need

- Dynamically-allocated variables
  - Do not go away until "deleted"
- If pointers are only private member data
  - They dynamically allocate "real" data
    - In constructor
  - Must have means to "deallocate" when object is destroyed
- Answer: destructor!

Destructors

- Opposite of constructor
  - Automatically called when object is out-of-scope
  - Default version only removes ordinary variables, not dynamic variables
- Defined like constructor, just add ~
  - MyClass::~MyClass()

```
{
  //Perform delete clean-up duties
}
```

Copy Constructors

- Automatically called when:
  1. Class object declared and initialized to other object
  2. When function returns class type object
  3. When argument of class type is "plugged in" as actual argument to call-by-value parameter
- Requires "temporary copy" of object
  - Copy constructor creates it
- Default copy constructor
  - Like default "=", performs member-wise copy
- Pointers → write own copy constructor!

Summary 1

- Pointer is memory address
  - Provides indirect reference to variable
- Dynamic variables
  - Created and destroyed while program runs
- Freestore
  - Memory storage for dynamic variables
- Dynamically allocated arrays
  - Size determined as program runs

Summary 2

- Class destructor
  - Special member function
  - Automatically destroys objects
- Copy constructor
  - Single argument member function
  - Called automatically when temp copy needed
- Assignment operator
  - Must be overloaded as member function
  - Returns reference for chaining