C++ Primer II

CMSC 202

Topics Covered

- Expressions, statements, blocks
- Control flow: if/else-if/else, while, do-while, for, switch
- Booleans, and non-bools as bools
- Preprocessor directives—intro
  (design aspects later)
- Functions (simple intro)

2

Topics Covered

More topics:

- Arrays
- Structures
- Basic pointers
- Comand-line arguments

3

Expressions

- An ***expression*** is a construct made up of variables, operators, and method invocations, that evaluates to a single value.
- For example:

```
int cadence = 0;
anArray[0] = 100;
cout << "Element 1 at index 0: " << anArray[0]);
int result = 1 + 2;
cout << (x == y ? "equal" :"not equal");
```

4

Statements

- ***Statements*** are roughly equivalent to sentences in natural languages. A ***statement*** forms a complete unit of execution.
- Two types of statements:
  - Expression statements – end with a semicolon ';'
    - Assignment expressions
    - Any use of ++ or --
    - Method invocations
    - Object creation expressions
  - Control Flow statements
    - Selection & repetition structures

5

If-Then Statement

- The *if-then* statement is the most basic of all the control flow statements.

Python

```
if x == 2:
    print "x is 2"
print "Finished"
```

C++

```
if (x == 2)
    cout << "x is 2";
cout << "Finished";
```

Notes about C++'s *if-then*:

- Conditional expression must be in parentheses
- Conditional expression has various interpretations of "truthiness" depending on type of expression

6

A few digressions, on:
- Multi-statement blocks
- Scope
- Truth in C++

7

Multiple Statements

- What if our *then* case contains multiple statements?

Python

```
if x == 2:
    print "even"
    print "prime"
print "Done!"
```

C++ *(but incorrect!!)*

```
if(x == 2)
    cout << "even";
    cout << "prime";
cout << "Done!";
```

Notes:
• Unlike Python, spacing plays no role in C++'s selection/repetition structures
• The C++ code is ***syntactically*** fine – no compiler errors
• However, it is ***logically*** incorrect

8

Blocks

- A ***block*** is a group of zero or more statements that are grouped together by delimiters.
- In C++, blocks are denoted by opening and closing curly braces '{' and '}' .

```
if(x == 2) {
    cout << "even";
    cout << "prime";
}
cout << "Done!";
```

Note:
• It is generally considered a good practice to include the curly braces even for single line statements.

9

Variable Scope

- You can define new variables in many places in your code, so where is it in effect?
- A variable's *scope* is the set of code statements in which the variable is known to the compiler.
- Where a variable can be referenced from in your program
- Limited to the code block in which the variable is defined
- For example:

```
if(age >= 18) {
    bool adult = true;
}
/* couldn't use adult here */
```

10

"Truthiness"**

- What is "true" in C++?
- Like some other languages, C++ has a true Boolean primitive type (*bool*), which can hold the constant values *true* and *false*
- Assigning a Boolean value to an *int* variable will assign 0 for *false,* 1 for *true*

** kudos to Stephen Colbert

11

"Truthiness"

- For compatibility with C, C++ is very liberal about what it allows in places where Boolean values are called for:
  - *bool* constants, variables, and espressions have the obvious interpretation
  - Any other integer-valued type is also allowed, and 0 is interpreted as "false", all other values as "false"
    - So, even -1 is considered true!

12

Gotcha! = versus ==

```
int a = 0;

if (a = 1) {
    printf ("a is one\n") ;
}
```

13

... and back to control flow structures

14

If-Then-Else Statement

- The *if-then-else* statement looks much like it does in Python (aside from the parentheses and curly braces).

Python

```
if x % 2 == 1:
    print "odd"
else:
    print "even"
```

C++

```
if(x % 2 == 1) {
    cout << "odd";
} else {
    cout << "even";
}
```

15

If-Then-Else If-Then-Else Statement

- Again, very similar…

Python

```
if x < y:
    print "x < y"
elif x > y:
    print "x > y"
else:
    print "x == y"
```

C++

```
if (x < y) {
    cout << "x < y";
} else if (x > y) {
    cout << "x > y";
} else {
    cout << "x == y";
}
```

16

Switch Statement

- Unlike *if-then* and *if-then-else*, the *switch* statement allows for any number of possible execution paths.
- Works with any integer-based (e.g., *char, int, long*) or enumerated type (covered later)

17

Switch Statement

```
int cardValue = /* get value from somewhere */;
switch(cardValue) {
    case 1:
        cout << "Ace";
        break;
    case 11:
        cout << "Jack";
        break;
    case 12:
        cout << "Queen";
        break;
    case 13:
        cout << "King";
        break;
    default:
        cout << cardValue;
}
```

Notes:
- *break* statements are typically used to terminate each *case.*
- It is usually a good practice to include a *default* case.

18

Switch Statement

```
switch (month) {
    case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
        cout << "31 days";
        break;
    case 4: case 6: case 9: case 11:
        cout << "30 days";
        break;
    case 2:
        cout << "28 or 29 days";
        break;
    default:
        cout << "Invalid month!";
        break;
}
```

Note:
- Without a break statement, cases “fall through” to the next statement.

19

Switch Statement

- To repeat: the switching value must evaluate to an integer or enumerated type (some other esoteric class types also allowed—not covered in class)
- The *case* values must be constant or literal, or enum value
- The case values must be of the same type as the switch expression

20

While Loops

- The *while* loop executes a block of statements while a particular condition is *true.*
- Pretty much the same as Python…

Python

```
count = 0;
while(count < 10):
    print count
    count += 1
print "Done!"
```

C++

```
int count = 0;
while(count < 10) {
    cout << count;
    count++;
}
cout << "Done!";
```

21

Do-While Loops

- In addition to *while* loops, Java also provides a *do-while* loop.
  - The conditional expression is at the bottom of the loop.
  - Statements within the block are always executed at least once.
  - Note the trailing semicolon!

```
int count = 0;
do {
    cout << count;
    count++;
} while(count < 10);
cout << "Done!";
```

22

For Loop

- The for statement provides a compact way to iterate over a range of values.

```
for (initialization; termination; increment) {
    /* ... statement(s) ... */
}
```

- The ***initialization expression*** initializes the loop – it is executed once, as the loop begins.
- When the ***termination expression*** evaluates to false, the loop terminates.
- The ***increment expression*** is invoked after each iteration through the loop.

23

For Loop

- The equivalent loop written as a *for* loop
  - Counting from start value (zero) up to (excluding) some number (10)

Python

```
for count in range(0, 10):
    print count
print "Done!"
```

C++

```
for (int count = 0; count < 10; count++) {
    cout << count;
}
cout << "Done!";
```

24

For Loop

- Counting from 25 up to (excluding) 50 in steps of 5

Python

```
for count in range(25, 50, 5):
    print count
print "Done!"
```

C++

```
for (int count = 25; count < 50; count += 5){
    cout << count;
}
cout << "Done!";
```

25

Range-based *for* Loop

- C++ has an equivalent for Python's *for-in* loop, or Java's "enhanced for" loop
- We will cover this alternate *for* form later when we learn about iterators

26

The *break* Statement

- The **break** statement can be used in **while**, **do-while**, and **for** loops to cause premature exit of the loop.
- THIS IS ***NOT*** A RECOMMENDED CODING TECHNIQUE.

27

Example break in a for Loop

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

int main( ) {
   int i;

   for (i = 1; i < 10; i++) {
     if (i == 5) {
        break;
      }
      cout << i << " ";
   }
   cout << "\nBroke out of loop at i = "
  << i;
   return 0 ;
}
```

<u>OUTPUT:</u>

1 2 3 4

Broke out of loop at i = 5.

28

The *continue* Statement

- The **continue** statement can be used in **while**, **do-while**, and **for** loops.
- It causes the remaining statements in the body of the loop to be skipped for the current iteration of the loop.
- THIS IS ***NOT*** A RECOMMENDED CODING TECHNIQUE.

29

Example continue in a for Loop

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

int main( ) {
   int i;

   for (i = 1; i < 10; i++) {
      if (i == 5) {
         continue;
      }
      cout << i << " ";
   }
   cout << "\nDone.\n";
   return 0 ;
}
```

<u>OUTPUT:</u>

1 2 3 4 6 7 8 9

Done.

30

Preprocessor Directives

31

Preprocessor Directives

- The C/C++ compiler has a preprocessing stage, called "cpp"
- It looks for *preprocessor directives:* lines beginning with a '#' (long before Twitter! ☺)
- These cause it to load other files, to define substitution macros that apply to the rest of the file being preprocessed, and other text-based modifications to your source
- An important point: your source file itself is not actually modified—the modifications are just done to a temporary copy at compile time

32

Preprocessor Directives

- Important directive 1: #include
  - This directive causes the preprocessor to insert, in place of this line, the entire contents of the specified file.
  - E.g.:

```
#include "myClass.h"
```

    Will cause the contents of the file "myClass.h" to be inserted into your source file at this point.
  - Note: no ';' at the end of this directive!

33

Preprocessor Directives

- Important directive 2: #define
  - This directive causes the preprocessor to add a macro definition into its internal table. After this, any appearance of the macro symbol in your source will be replaced with the macro expansion
  - E.g.:

```
#define PI 3.14159
…
int area = radius * PI;
```

This will be exactly equivalent to:

```
int area = radius * 3.14159;
```

- NB: there is no '=', nor any ';' in the definition!

34

Basic Functions

35

Predefined Functions

- C++ has standard libraries full of functions for our use!
- Must "#include" appropriate library
  - e.g.,
    - <cmath>, <cstdlib> (Original "C" libraries)
    - <iostream> (for cout, cin)

3-36

The Function Call

- Sample function call and result assignment:
  theRoot = sqrt(9.0);
  - The expression "sqrt(9.0)" is known as a function *call*, or function *invocation*
  - The argument in a function call (9.0) can be a literal, a variable, or a complex expression
  - A function can have an arbitrary number of arguments
  - The call itself can be part of an expression:
    - bonus = sqrt(sales * commissionRate)/10;
    - A function call is allowed wherever it's legal to use an expression of the function's return type

3-37

More Predefined Functions

- #include <cstdlib>
  - Library contains functions like:
    - abs() // Returns absolute value of an int
    - labs() // Returns absolute value of a long int
    - *fabs() // Returns absolute value of a float
  - *fabs() is actually in library <cmath>!
    - Can be confusing
    - Remember: libraries were added after C++ was "born," in incremental phases
    - Refer to appendices/manuals for details

3-38

Even More Math Functions: **Display 3.2** Some Predefined Functions (1 of 2)

Display 3.2 **Some Predefined Functions**

NAME	DESCRIPTION	TYPE OF ARGUMENTS	TYPE OF VALUE RETURNED	EXAMPLE	VALUE	LIBRARY HEADER
sqrt	Square root	double	double	sqrt(4.0)	2.0	cmath
pow	Powers	double	double	pow(2.0,3.0)	8.0	cmath
abs	Absolute value for int	int	int	abs(-7) abs(7)	7 7	cstdlib
labs	Absolute value for long	long	long	labs(-70000) labs(70000)	70000 70000	cstdlib
fabs	Absolute value for double	double	double	fabs(-7.5) fabs(7.5)	7.5 7.5	cmath

3-39

Even More Math Functions:
Display 3.2 Some Predefined Functions (2 of 2)

ceil	Ceiling (round up)	double	double	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	Floor (round down)	double	double	floor(3.2) floor(3.9)	3.0 3.0	cmath
exit	End program	int	void	exit(1);	None	cstdlib
rand	Random number	None	int	rand( )	Varies	cstdlib
srand	Set seed for rand	unsigned int	void	srand(42);	None	cstdlib

Random Number Generator

- Return "randomly chosen" number
- Used for simulations, games
  - rand()
    - Takes no arguments
    - Returns value between 0 & RAND_MAX
  - Scaling
    - Squeezes random number into smaller range
      rand() % 6
    - Returns random value between 0 & 5
  - Shifting
    rand() % 6 + 1
    - Shifts range between 1 & 6 (e.g., die roll)

Random Number Seed

- Pseudorandom numbers
  - Calls to rand() produce given "sequence" of random numbers
- Use "seed" to alter sequence
  srand(seed_value);
  - void function
  - Receives one argument, the "seed"
  - Can use any seed value, including system time:
    srand(time(0));
  - time() returns system time as numeric value
  - Library <time> contains time() functions

Random Examples

- Random double between 0.0 & 1.0:
  (RAND_MAX – rand())/static_cast<double>(RAND_MAX)
  – Type cast used to force double-precision division
- Random int between 1 & 6:
  rand() % 6 + 1
  – "%" is modulus operator (remainder)
- Random int between 10 & 20:
  rand() % 10 + 10

3-43

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

3-44

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

3-45

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
- Detail: parameter types are manadatory, but names are optional

3-46

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);
}
```

3-47

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

3-48

Function Call

- Just like calling predefined function
  bill = totalCost(number, price);
- Recall: 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

3-49

Function Example:
Display 3.5 A Function to Calculate Total Cost (1 of 2)

Display 3.5

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

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

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

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

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

Function declaration; also called the function prototype

Function call

3-50

Function Example:
Display 3.5 A Function to Calculate Total Cost (1 of 2)

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

    return 0;
}

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

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

Function head

Function body

Function definition

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

3-51

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

3-52

Declaring Void Functions

- "void" functions are called for side effects; don't return any usable value
- Declaration is similar to functions returning a value, but return type specified as "void"
- Example:
  - Function declaration/prototype:

```
void showResults(double fDegrees,
                 double cDegrees);
```

    - Return-type is "void"
    - Nothing is returned

3-53

More on Return Statements

- Transfers control back to "calling" function
  - For return type other than void, MUST have return statement
  - Typically the LAST statement in function definition
- return statement optional for void functions
  - Closing } would implicitly return control from void function

3-54

Preconditions and Postconditions

- Similar to "I-P-O" discussion
- Comment function declaration:

```
void showInterest(double balance, double rate);
//Precondition: balance is nonnegative account balance
//          rate is interest rate as percentage
//Postcondition: amount of interest on given balance,
//          at given rate …
```

- Often called Inputs & Outputs

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"

Scope Rules

- Local variables
  - Declared inside body of given function
  - Available only within that function
- Can have variables with same names declared in different functions
  - Scope is local: "that function is it's scope"
- Local variables preferred
  - Maintain individual control over data
  - Need to know basis
  - Functions should declare whatever local data needed to "do their job"

Global Constants and Global Variables

- Declared "outside" function body
  - Global to all functions in that file
- Declared "inside" function body
  - Local to that function
- Global declarations typical for constants:
  - const double TAXRATE = 0.05;
  - Declare globally so all functions have scope
- Global variables?
  - Possible, but SELDOM-USED
  - Dangerous: no control over usage!

3-58

Blocks

- Declare data inside compound statement
  - Called a "block"
  - Has "block-scope"
- Note: all function definitions are blocks!
  - This provides local "function-scope"
- Loop blocks:

```
for (int ctr=0;ctr<10;ctr++)
{
 sum+=ctr;
}
```

  - Variable ctr has scope in loop body block only

3-59

Nested Scope

- Same name variables declared in multiple blocks
- Very legal; scope is "block-scope"
  - No ambiguity
  - Each name is distinct within its scope

3-60

Arrays

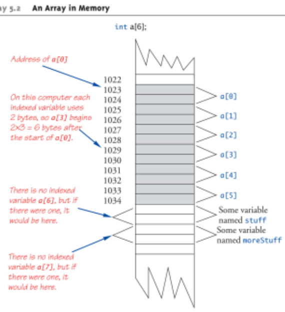
Arrays

- An array is an aggregate (grouping under a common name) of related data items that all have the same data type
- Arrays can be of any data type we choose.
- Arrays are **static** in that they remain the same size throughout program execution.
- An array's data items are stored contiguously in memory
- To declare an array of 5 integers called "numbers", you would use:

```
int numbers[5];
```

An Array in Memory

Display 5.2 **An Array in Memory**



Array Declaration and Initialization

```
int numbers[5];
```

- This declaration sets aside a chunk of memory that is big enough to hold 5 integers.
- It does not initialize those memory locations to 0 or any other value. They contain garbage.
- Initializing an array may be done with an **array initializer**, as in :
  int numbers[5] = { 5, 2, 6, 9, 3 } ;

numbers →

5	2	6	9	3

Auto-Initializing Arrays

- If fewer values than size supplied:
  - Fills from beginning
  - Fills "rest" with zero of array base type
- If array-size is left out
  - Declares array with size required based on number of initialization values
  - Example:
    int b[] = {5, 12, 11};
    - Allocates array b to size 3

5-65

Array Declaration and Initialization

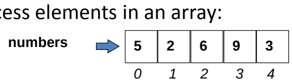
- A special case is an "array of chars":
  char name[5] ;
- As mentioned earlier, a C-string is in fact an array of chars, usually ending in a 0
  - The 0-valued char at the end is called a "null terminator"
  - Strings do not necessarily have to be null-terminated.
- Initializing a char array may be done the usual way, as in:
  `char name[5] = { 'J', 'o', 'h', 'n', 0 };`
  …or with a string constant:
  `char name[5] = "John" ;`

name →

'J'	'o'	'h'	'n'	'\0'

Accessing Array Elements

- You use the standard bracketed subscript notation to access elements in an array:




```
cout << "The third element is " << numbers[2];
```

would give the output

The third element is 6

- Subscripts are integers and always begin at zero.

Accessing Array Elements (con't)

- A subscript can also be any expression that evaluates to an integer.

  numbers[(a + b) * 2] ;

- Caution! C++ does not do bounds checking for simple arrays, so you must ensure you are staying within bounds

Defined Constant as Array Size

- Always use defined/named constant for array size
- Example:
  const int NUMBER_OF_STUDENTS = 5;
  int score[NUMBER_OF_STUDENTS];
- Improves readability
- Improves versatility
- Improves maintainability

5-69

Arrays in Functions

- As arguments to functions
  - Indexed variables
    - An individual "element" of an array can be function parameter
  - Entire arrays
    - All array elements can be passed as "one entity"
- As return value from function
  - Can be done → chapter 10

5-70

Indexed Variables as Arguments

- Indexed variable handled same as simple variable of array base type
- Given this function declaration:
  void myFunction(double par1);
- And these declarations:
  int i; double n, a[10];
- Can make these function calls:
  myFunction(i); // i is converted to double
  myFunction(a[3]); // a[3] is double
  myFunction(n); // n is double

5-71

Entire Arrays as Arguments

- Formal parameter can be entire array
  - Argument then passed in function call is array name
  - Called "array parameter"
- Send size of array as well
  - Typically done as second parameter
  - Simple int type formal parameter

5-72

Entire Array as Argument Example: **Display 5.3** Function with an Array Parameter

Display 5.3 **Function with an Array Parameter**

SAMPLE DIALOGUEFUNCTION DECLARATION

```
void fillUp(int a[], int size);
//Precondition: size is the declared size of the array a.
//The user will type in size integers.
//Postcondition: The array a is filled with size integers
//from the keyboard.
```

SAMPLE DIALOGUEFUNCTION DEFINITION

```
void fillUp(int a[], int size)
{
    cout << "Enter " << size << " numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> a[i];
    cout << "The last array index used is " << (size - 1) << endl;
}
```


Entire Array as Argument Example

- Given previous example:
- In some main() function definition, consider this call:
  int score[5], numberOfScores = 5;
  fillUp(score, numberOfScores);
    - 1st argument is entire array
    - 2nd argument is integer value
  - Note no brackets in array argument!

Array as Argument: How?

- What's really passed?
- Think of array as 3 "pieces"
  - Address of first indexed variable (arrName[0])
  - Array base type
  - Size of array
- Only 1st piece is passed!
  - Just the beginning address of array
  - Very similar to "pass-by-reference"

Array Parameters

- May seem strange
  - No brackets in array argument
  - Must send size separately
- One nice property:
  - Can use SAME function to fill any size array!
  - Exemplifies "re-use" properties of functions
  - Example:
```
int score[5], time[10];
fillUp(score, 5);
fillUp(time, 10);
```


Multidimensional Arrays

- Arrays with more than one index
  - char page[30][100];
    - Two indexes: An "array of arrays"
    - Visualize as:
```
page[0][0], page[0][1], …, page[0][99]
page[1][0], page[1][1], …, page[1][99]
…
page[29][0], page[29][1], …, page[29][99]
```
- C++ allows any number of indexes
  - Typically no more than two

Multidimensional Array Parameters

- Similar to one-dimensional array
  - 1st dimension size not given
    - Provided as second parameter
  - 2nd dimension size IS given
- Example:
```
void DisplayPage(const char p[][100], int sizeDimension1)
{
 for (int index1=0; index1<sizeDimension1; index1++)
 {
     for (int index2=0; index2 < 100; index2++)
         cout << p[index1][index2];
     cout << endl;
 }
}
```

Array Limitations

- Simple arrays have limitations
  - Array out-of-bounds access
  - No resizing
  - Hard to get current size
  - Not initialized
  - Much of this Is due to issues of efficiency and backwards-compatibility, which are high priorities in C/C++
- Later, we will learn about the *vector* class, which addresses many of these issues

81

Structures

82

Structures

- 2nd aggregate data type: struct
- Recall: aggregate meaning "grouping"
  - Recall array: collection of values of same type
  - Structure: collection of values of different types
- Treated as a single item, like arrays
- Major difference: Must first "define" struct
  - Prior to declaring any variables

6-83

Structure Types

- Define struct globally (typically)
- No memory is allocated
  - Just a "placeholder" for what our struct will "look like"
- Definition:

```
struct CDAccountV1  ←Name of new struct "type"
{
    double balance;   ← member names
    double interestRate;
    int term;
};
```

6-84

Declare Structure Variable

- With structure type defined, now declare variables of this new type:
  CDAccountV1 account;
  - Just like declaring simple types
  - Variable *account* now of type CDAccountV1
  - It contains "member values"
    - Each of the struct "parts"

6-85

Accessing Structure Members

- Dot Operator to access members
  - account.balance
  - account.interestRate
  - account.term
- Called "member variables"
  - The "parts" of the structure variable
  - Different structs can have same name member variables
    - No conflicts

6-86

struct Example

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

struct Account {
    double balance;
    double interestRate;
    int    term;
};

Account getAccountInfo(void);
void printAccountInfo(Account acct);

int main(int argc, char *argv[]) {
    Account myAcct;

    myAcct = getAccountInfo();
    printAccountInfo(myAcct);
    return 0;
}
```

struct Example (cont)

```
Account getAccountInfo(void) {
    Account acct;

    cout << "Enter balance: ";
    cin >> acct.balance;
    cout << "Enter interest rate: ";
    cin >> acct.interestRate;
    cout << "Enter account term: ";
    cin >> acct.ter,;
    return acct;
}

void printAccountInfo(Account acct) {
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Your balance is $" << acct.balance
         << ", with an interest rate of " << acct.interestRate
         << "\nThe term is " << account.term << " months\n";
}
```

Structure Pitfall

- Semicolon after structure definition
  - ; MUST exist:
    struct WeatherData
    {
      double temperature;
      double windVelocity;
    }; ← REQUIRED semicolon!
  - Required since you "can" declare structure variables in this location

6-92

Structure Assignments

- Given structure named CropYield
- Declare two structure variables:
  CropYield apples, oranges;
  - Both are variables of "struct type CropYield"
  - Simple assignments are legal:
    apples = oranges;
    - Simply copies each member variable from apples into member variables from oranges

6-93

Structures as Function Arguments

- Passed like any simple data type
  - Pass-by-value
  - Pass-by-reference (covered later)
  - Or combination
- Can also be returned by function
  - Return-type is structure type
  - Return statement in function definition sends structure variable back to caller

6-94

Initializing Structures

- Can initialize at declaration
  - Example:
    ```
    struct Date
    {
        int month;
        int day;
        int year;
    };
    Date dueDate = {12, 31, 2003};
    ```
  - Declaration provides initial data to all three member variables

6-95

Structures vs Classes

- Structures have existed since the early days of C, due to the importance of being able to aggregate heterogeneous data about a single entity
- OOP is a natural follow-on to structured data
- So, in a manner of speaking, classes supersedes structures

96

Basic Pointers

97

Basic Pointers

- Early on in computer and programming language design, it was found important to be able to flexibly address different variables under program control (think about addressing array elements with variable indices)
- Computer architects added ability to do “indirect” addressing: taking a variable value (i.e., memory location contents) and applying that value as the numerical location of another variable

98

Basic Pointers

- We can do this in both directions:
  - Put a numerical value into a variable (i.e., memory location) and tell the processor to do an operation on the value in the location *addressed* by the first value
  - Given a variable (again, a memory location), take it's memory address in RAM, which is a number, and store this number inside some other variable
    - This requires the cooperation of the compiler, which decides, and therefore knows, where the various variables are being stored in RAM.

Pointer Introduction

- Pointer definition:
  - A variable holding memory address of another variable
  - an expression evaluating to such a value
- We use the '*' ("points to") and '&' ("address of") unary operators to work with pointers
- Note distinction between a pointer--which is a numerical address and therefore always a certain size (number of bytes) on a given computer—and the type of data it "points to", which can be of different sizes

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

10-102

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

10-103

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

10-104

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

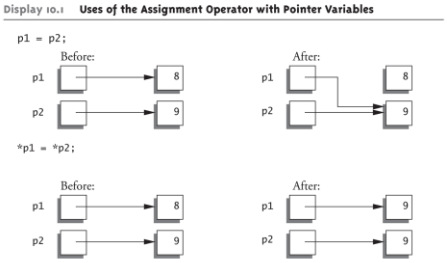
- The "address of" operator
- Also used to specify call-by-reference parameter (more on this later)
  - No coincidence!
  - Recall: call-by-reference parameters pass "address of" the actual argument
- Operator’s two uses are closely related

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**



Pointer Math

- Why is a pointer != an integer? Consider:

```
int numbers[50];
int *numPtr;

numPtr = numbers[25];   // NOT LEGAL!!!
numPtr = &numbers[25];  // Okay
*numPtr = 47;  // same as "numbers[25] = 47"
numPtr += 1;
// Above adds 4(!), not 1, to value of numPtr,
// since the thing numPtr points to (an int)
// occupies 4 bytes
```

111

Simulated "Pass by Reference"

- Some programming languages provide mechanism for called function to have direct access to variables used in the calling function
- We can simulate this by using pointers (see following slide)
- C++ added true "call by reference" – we will see this later on

112

Simulated "Pass by Reference"

- Calling function:

```
int x = 1;

// pass in reference to (actually, pointer to)
// our argument variable "x"
add1(&x);
cout << x;  // will output 2!
```

- Called function:

```
void add1(int *var) {
    *var = *var + 1;
}
```

113

Command-Line Arguments

114

Command-Line Arguments

```
int main (int argc, char *args[]){
        for(int i = 0; i < argc; i++){
            cout << "Arg " << i << ": " << argv[i] << endl;
        }
    }
}
```

- Anything in the shell command line (including the name of the program to be executed) will be read as a ***command line argument***.
- All text entered will be stored in the C-string array specified in main (typically ***argv*** by convention).
  - myprog.out Hi
  - Results in "myprog.out" stored at argv[0], and "Hi" stored at argv[1]
- Individual arguments can be separated by spaces like so
  - myprog.out foo 123 bar
  - Results in "foo" stored at argv[1], "123" at argv[2] and "bar" at argv[3]

115