CS 31: Introduction To Computer Science I

Howard A. Stahl

Agenda

- `void` Functions
- `return` Statement
- PostFix and Prefix ++, -- Operators
- Default Value Arguments
- Parameter Passing Mechanisms
- Problem Solving and Testing Strategy

`void` Functions

- Functions Need Not Always Return A Result
- A Return Type Of `void` Indicates A Function That Returns No Result
  - `return` statement contains no expression
  - `return` statement assumed at end of function
- In Other Languages, `void` Functions Are Called Subroutines

`void` Functions

- Examples:

```
void horizontal_line( ) {
 cout << "\n-------------\n";
 return;
}
void sayGoodnightGracie( ) {
 cout << "Goodnight, Gracie";
}
```

`void` Functions

- Examples:

```
void horizontal_line( ) {
 cout << "\n-------------\n";
 return;
}
void sayGoodnightGracie( ) {
 cout << "Goodnight, Gracie";
}
```

return statement is assumed

`return` Statement

- A Function May Contain Multiple `return` Statements

```
int max( int a, int b) {
    if (a < b)
      return b;
    else
      return a;
}
```

- Generally, More Readable With Just One

Prefix and Postfix Operators

- `++` Is A Shorthand For `+ 1`
  - `i++;` $\longrightarrow$ `i = i + 1;`
- `--` Is A Shorthand For `- 1`
  - `i--;` $\longrightarrow$ `i = i - 1;`
- The Operator Can Come Before Or After The Variable
  - `i++;      ++i;`

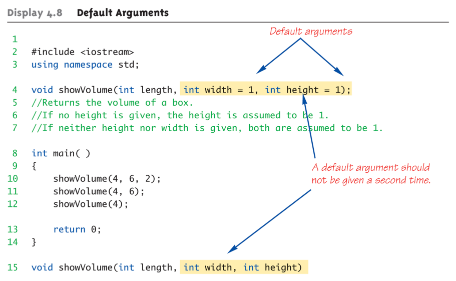
Prefix and Postfix Operators

- Prefix Operator Occurs Before Expression Evaluation
- Postfix Operator Occurs After Statement Evaluation
  - `int i = 12, j = 10, k = 0;`
  - `k = i++ * --j;`
  - `k = --i + ++j;`

Default Valued Arguments

- Functions Can Have "Optional" Arguments
- They Are Defined, But Do No Need To Be Passed By The Caller

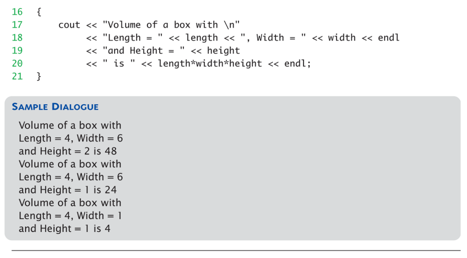
Default Value Arguments



Display 4.8 **Default Arguments**

```

#include <iostream>
using namespace std;

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

int main( )
{
    showVolume(4, 6, 2);
    showVolume(4, 6);
    showVolume(4);

    return 0;
}

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

Default Value Arguments



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

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
```

Default Valued Arguments

- Functions Can Have “Optional” Arguments
- They Are Defined, But Do No Need To Be Passed By The Caller
- If Not Passed, A Default Value Will Be Supplied Automagically
- Default Valued Arguments Must Be Grouped Together At The End Of The Parameter List

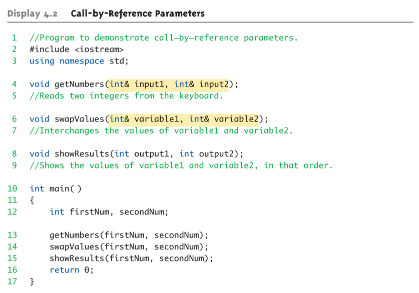
Parameter Passing

- So Far, Our Functions Cannot Alter Their Parameters
  - referred to as "pass-by-value"
  - these functions can only provide a single output value
- However, There Is Another Kind Of Parameter Passing Scheme
  - referred to as "pass-by-reference"

Parameter Passing

- Reference Parameters Are Not Copies Of The Actual Parameter, But Are The Parameters Themselves
- Actual Parameters Must Be A Variable
  - referred to as an "lvalue", as opposed to an "rvalue"
- Specified When The Prototype Use The Syntax: `type&`
  - recall from C that `&` means "address of"

Call-By-Reference Example



Display 4.2 Call-by-Reference Parameters

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

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

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

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

int main( )
{
    int firstNum, secondNum;

    getNumbers(firstNum, secondNum);
    swapValues(firstNum, secondNum);
    showResults(firstNum, secondNum);
    return 0;
}
```

Call-By-Reference Example

```
void getNumbers(int& input1, int& input2)
{
    cout << "Enter two integers: ";
    cin >> input1
        >> input2;
}

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

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}

void showResults(int output1, int output2)
{
    cout << "In reverse order the numbers are: "
         << output1 << " " << output2 << endl;
}
```

Call-By-Reference Example

Display 4.2 **Call-by-Reference Parameters**

SAMPLE DIALOGUE

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

Parameter Passing

- Reference Parameter Example:

```
void swap(int& x, int& y) {
  int temp = x;
  x = y;
  y = temp;
}
```

- Legal Invocation???

```
int i=0, j=20;
swap( i, j );
```

Parameter Passing

- Reference Parameter Example:

```
void swap(int& x, int& y) {
  int temp = x;
  x = y;
  y = temp;
}
```

- Legal Invocation???

```
int i=0, j=20;
swap( i, j++ );
```

Parameter Passing

- Reference Parameter Example:

```
void swap(int& x, int& y) {
  int temp = x;
  x = y;
  y = temp;
}
```

- Legal Invocation???

```
int i=0, j=20;
swap( 7-10, i/j );
```

Function Call And Return

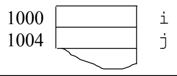
```
void swap( int &x, int &y);
main( )
```

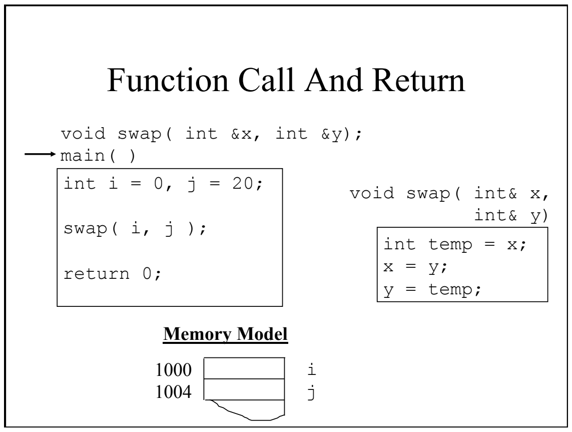
```
int i = 0, j = 20;

swap( i, j );

return 0;
```

```
void swap( int& x,
           int& y)
```

```
int temp = x;
x = y;
y = temp;
```

Memory Model





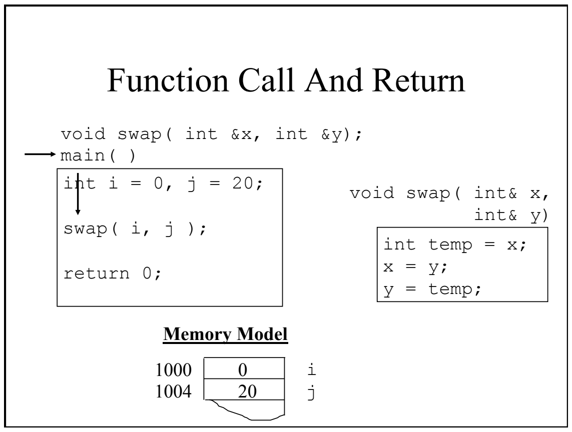
Function Call And Return

```
void swap( int &x, int &y);
main( )
int i = 0, j = 20;

swap( i, j );

return 0;
```

```
void swap( int& x,
           int& y)
int temp = x;
x = y;
y = temp;
```

Memory Model

1000		i
1004		j



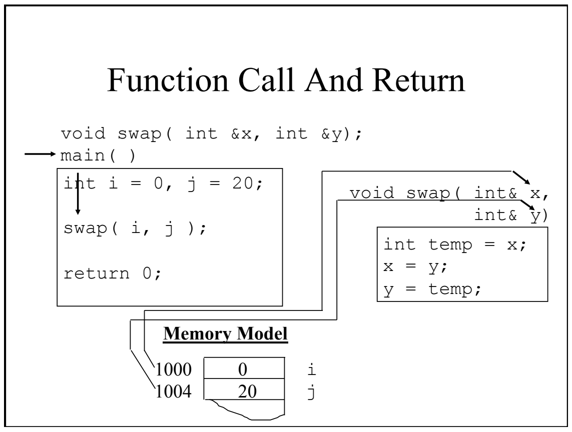
Function Call And Return

```
void swap( int &x, int &y);
main( )
int i = 0, j = 20;

swap( i, j );

return 0;
```

```
void swap( int& x,
           int& y)
int temp = x;
x = y;
y = temp;
```

Memory Model

1000	0	i
1004	20	j



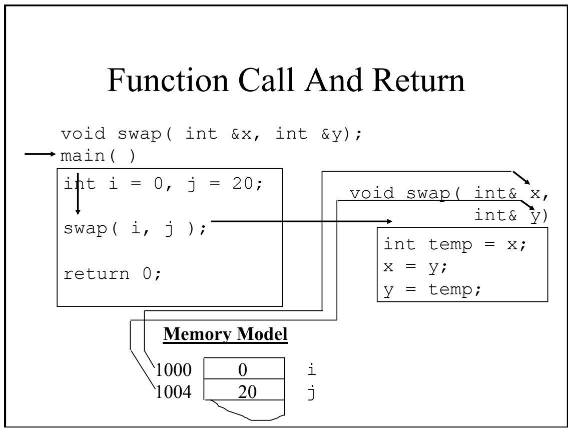
Function Call And Return

```
void swap( int &x, int &y);
main( )
int i = 0, j = 20;

swap( i, j );

return 0;
```

```
void swap( int& x,
           int& y)
int temp = x;
x = y;
y = temp;
```

Memory Model

1000	0	i
1004	20	j



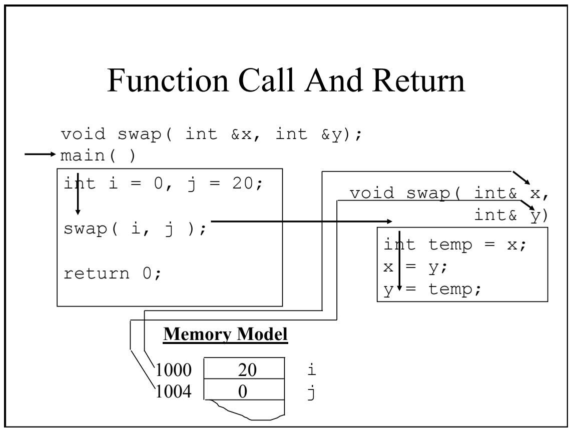
Function Call And Return

```
void swap( int &x, int &y);
main( )
int i = 0, j = 20;

swap( i, j );

return 0;
```

```
void swap( int& x,
           int& y)
int temp = x;
x = y;
y = temp;
```

Memory Model

1000	0	i
1004	20	j



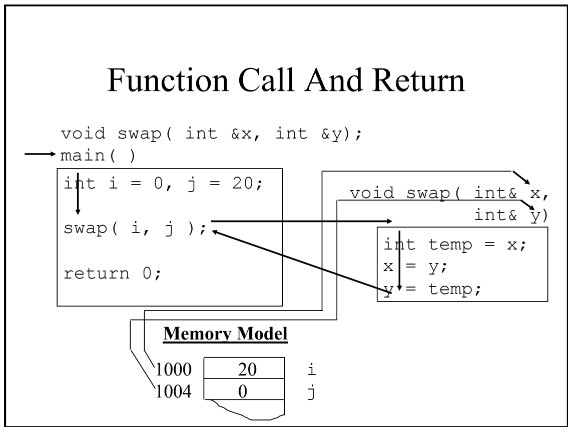
Function Call And Return

```
void swap( int &x, int &y);
main( )
int i = 0, j = 20;

swap( i, j );

return 0;
```

```
void swap( int& x,
           int& y)
int temp = x;
x = y;
y = temp;
```

Memory Model

1000	20	i
1004	0	j



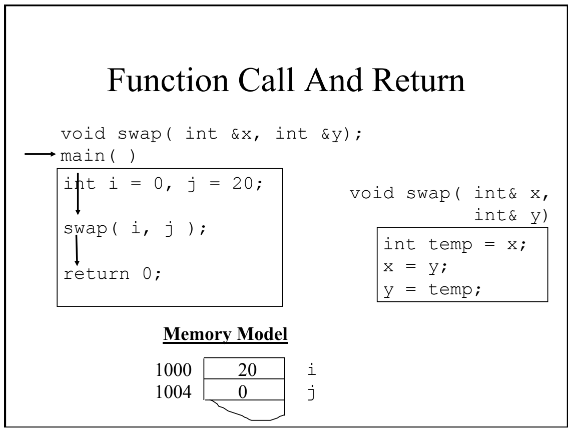
Function Call And Return

```
void swap( int &x, int &y);
main( )
int i = 0, j = 20;

swap( i, j );

return 0;
```

```
void swap( int& x,
           int& y)
int temp = x;
x = y;
y = temp;
```

Memory Model

1000	20	i
1004	0	j

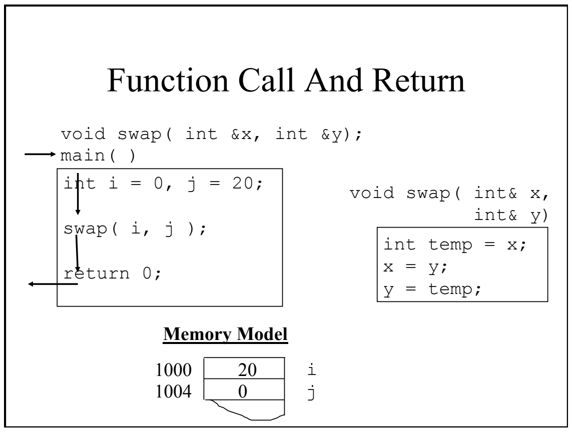


Function Call And Return

```
  void swap( int &x, int &y);
→ main( )
  int i = 0, j = 20;
                            void swap( int& x,
  swap( i, j );                        int& y)
                              int temp = x;
  return 0;                   x = y;
                              y = temp;
```

Memory Model

1000	20	i
1004	0	j

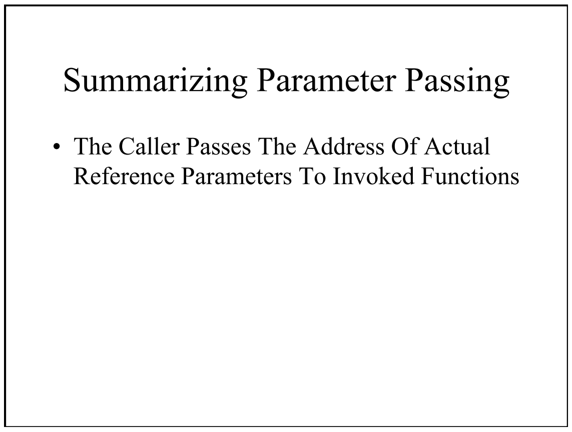


Function Call And Return

```
  void swap( int &x, int &y);
→ main( )
  int i = 0, j = 20;
                            void swap( int& x,
  swap( i, j );                        int& y)
                              int temp = x;
← return 0;                   x = y;
                              y = temp;
```

Memory Model

1000	20	i
1004	0	j



Summarizing Parameter Passing

- The Caller Passes The Address Of Actual Reference Parameters To Invoked Functions

Time For Our First Demo!

- Reference.cpp

(See Handout For Example 1)

Summarizing Our First Demo!

- Pass-By-Value Results In Copies Being Made Of Every Argument
  - this might have a performance impact on your code
- However, Pass-By-Reference Makes Things More Complex
  - your function may have unintended side effects, since it can change values inside the caller's world

Mixing Parameter Types

- A Function May Use Both Kinds Of Parameter Passing Schemes In One Prototype

```
void process( int input, int& output );
```

this parameter passed by value

this parameter passed by reference

Problem Solving Strategy

- One big problem is harder to solve than many smaller problems
- Understand the problem
  - what result is expected
  - what process can provide these results
  - what parameters are needed for these processes
  - write function descriptions in english telling what the function should do

Problem Solving Strategy

- C++ Syntax Typically Obscures Understanding
  - write out your solution on paper FIRST
  - use flow charts or pseudocode
  - translate to C++ syntax on paper
  - try not to compose code at a terminal
- Great Answers Don’t Come The First Time
  - iteratively refine and enhance partial solutions

Testing Strategy

- How Do You Test Functions?
  - Test One Function At A Time
  - Display Intermediate Results
  - You May Need To Create Test Data To Use Via “Driver Programs”
  - If The Function Being Tested Calls Other Functions, Create “Stubs”
  - Try Varying One Thing At A Time
    - if something goes wrong, you know what changed

Testing Strategy

- Drivers
  - allows you to test a function without all the rest of a program
  - just to execute the function and show its results
  - often, provides a loop to retest the function on different arguments

Testing Strategy

- Stubs
  - simplified version of a function not written or tested yet
  - often used when testing another function
  - does not necessarily deliver correct values
  - works best when stubs are replaced by actual functions, one at a time

Time For Our Next Demo!

- TestDriver.cpp

(See Handout For Example 2)

Summarizing Our Second Demo!

- Drivers Are Throwaway Code Meant To Exercise Other Code
- Stubs Are Fake StandIns For Code That Will Be Fleshed Out Later

An assert Macro

- Useful In Debugging
- Stops Execution So Problems Can Be Corrected

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 < coinValue ) && (coinValue < 100) && (0 <= amountLeft) && (amountLeft< 100));`
  - If precondition not satisfied → condition is false → program execution terminates!

assert On/Off

- Preprocessor Provides This For Us To Use

-
  ```
  #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

Summary

- `void` Functions
- `return` Statement
- PostFix and Prefix ++, -- Operators
- Default Value Arguments
- Parameter Passing Mechanisms
- Problem Solving and Testing Strategy