Static Local Variables

- Local variables only exist while the function is executing. When the function terminates, the contents of local variables are lost.
- static local variables retain their contents between function calls.
- static local variables are defined and initialized only the first time the function is executed. 0 is the default initialization value.

Program 6-21

```
// This program shows that local variables do not retain
// their values between function calls.
#include <iostream>
using namespace std;

// Function prototype
void showLocal();

int main()
{
   showLocal();
   showLocal();
   return 0;
}

```

(Program Continues)

Program 6-21 *(continued)*

```
//*************************************************************
// Definition of function showLocal.                          *
// The initial value of localNum, which is 5, is displayed.   *
// The value of localNum is then changed to 99 before the     *
// function returns.                                          *
//*************************************************************

void showLocal()
{
   int localNum = 5; // Local variable

   cout << "localNum is " << localNum << endl;
   localNum = 99;
}
```

Program Output

```
localNum is 5
localNum is 5
```

In this program, each time `showLocal` is called, the `localNum` variable is re-created and initialized with the value 5.

A Different Approach, Using a Static Variable

Program 6-22

```
// This program uses a static local variable.
#include <iostream>
using namespace std;

void showStatic(); // Function prototype

int main()
{
   // Call the showStatic function five times.
   for (int count = 0; count < 5; count++)
      showStatic();
   return 0;
}

```

(Program Continues)

Program 6-22 *(continued)*

```
//*************************************************************
// Definition of function showStatic.                         *
// statNum is a static local variable. Its value is displayed *
// and then incremented just before the function returns.     *
//*************************************************************

void showStatic()
{
   static int statNum;

   cout << "statNum is " << statNum << endl;
   statNum++;
}
```

Program Output

```
statNum is 0
statNum is 1
statNum is 2
statNum is 3
statNum is 4
```

`statNum` is automatically initialized to 0. Notice that it retains its value between function calls.

If you do initialize a local static variable, the initialization only happens once. See Program 6-23.

```
//*************************************************************
// Definition of function showStatic.                         *
// statNum is a static local variable. Its value is displayed *
// and then incremented just before the function returns.     *
//*************************************************************

void showStatic()
{
   static int statNum = 5;

   cout << "statNum is " << statNum << endl;
   statNum++;
}
```

Program Output

```
statNum is 5
statNum is 6
statNum is 7
statNum is 8
statNum is 9
```

Default Arguments

- A <u>Default argument</u> is an argument that is passed automatically to a parameter if the argument is missing on the function call.

- Must be a constant declared in prototype:

```
void evenOrOdd(int = 0);
```

- Can be declared in header if no prototype

- Multi-parameter functions may have default arguments for some or all of them:

```
int getSum(int, int=0, int=0);
```

Default Arguments

- If not all parameters to a function have default values, the defaultless ones are declared first in the parameter list:

```
int getSum(int, int=0, int=0);// OK
int getSum(int, int=0, int);  // NO
```

- When an argument is omitted from a function call, all arguments after it must also be omitted:

```
sum = getSum(num1, num2);     // OK
sum = getSum(num1, , num3);   // NO
```

Call-By-Value

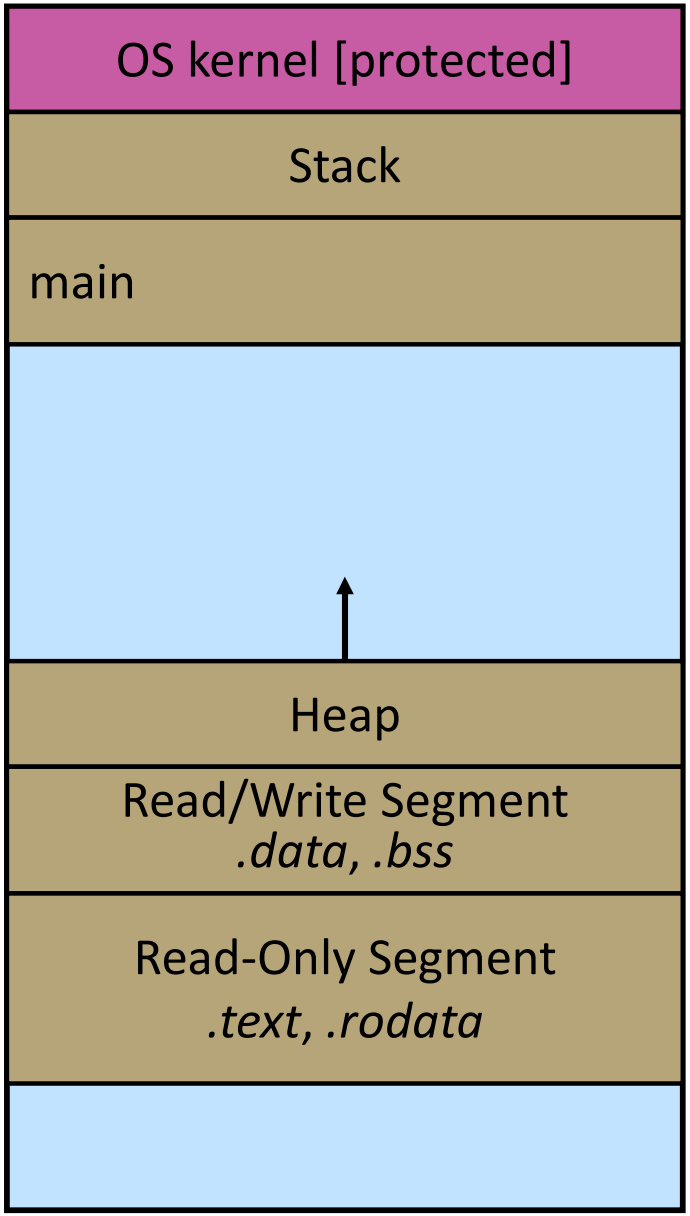
- C++ pass arguments by *value*
  - Callee receives a **local copy** of the argument
    - Register or Stack
  - If the callee modifies a parameter, the caller's copy *isn't* modified

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```

Broken Swap

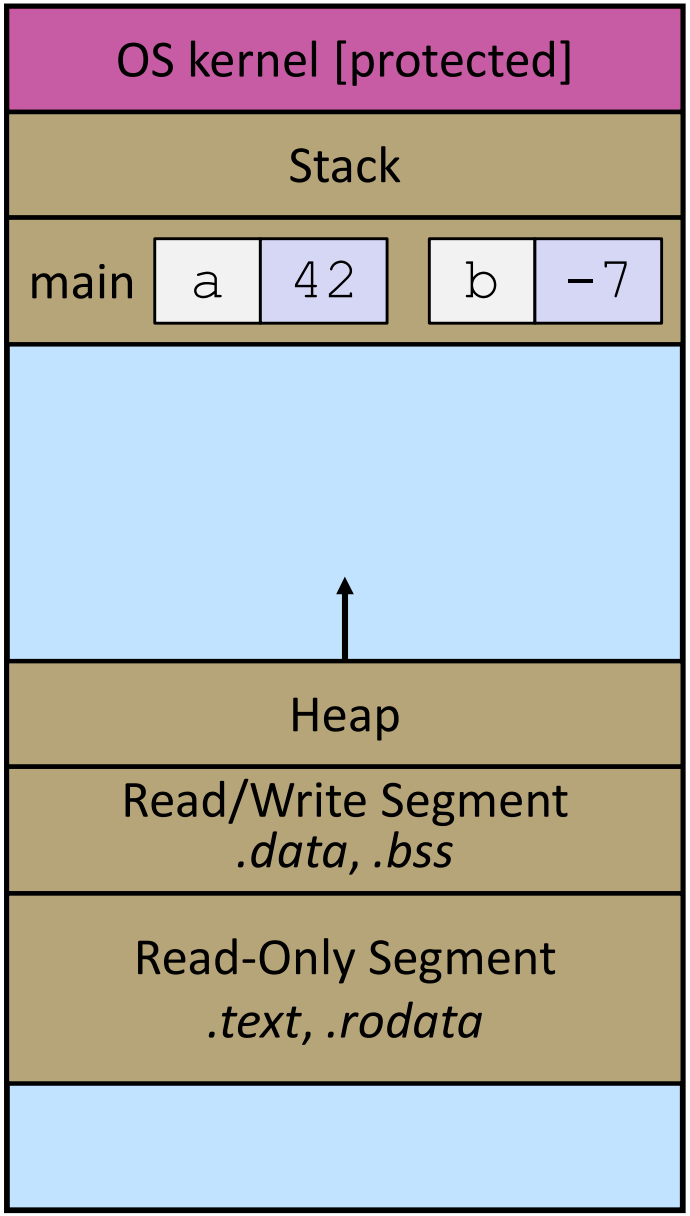
Note: Arrow points to *next* instruction.

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```



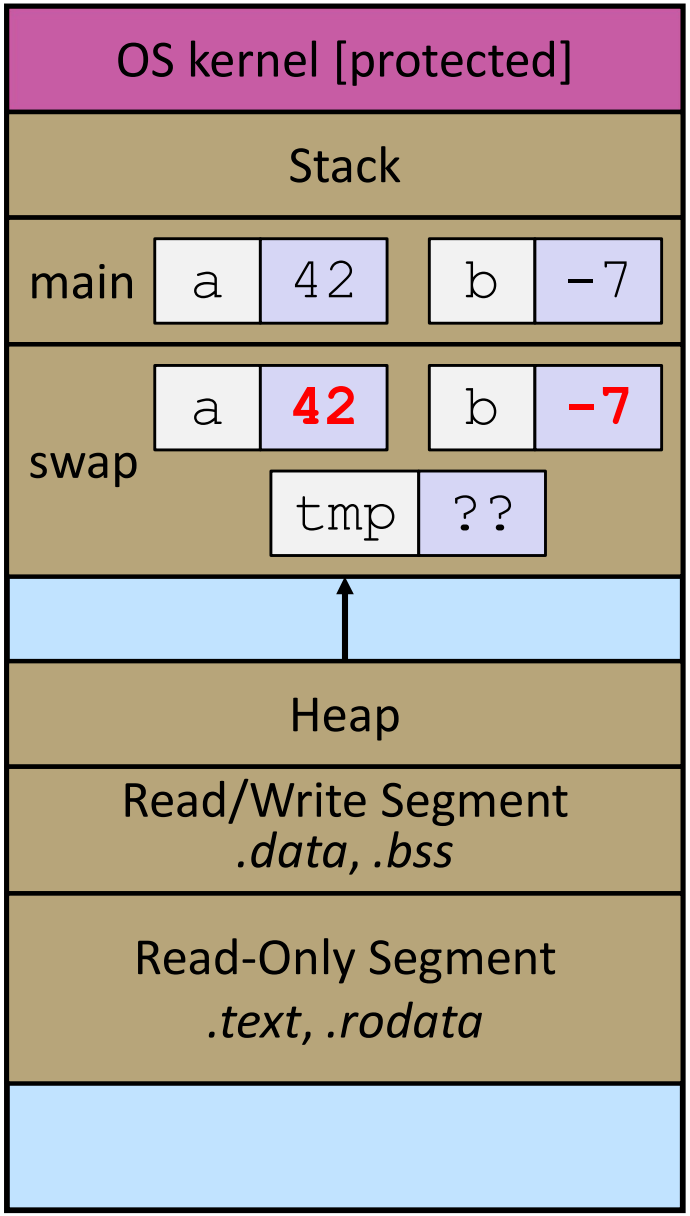
Broken Swap

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```



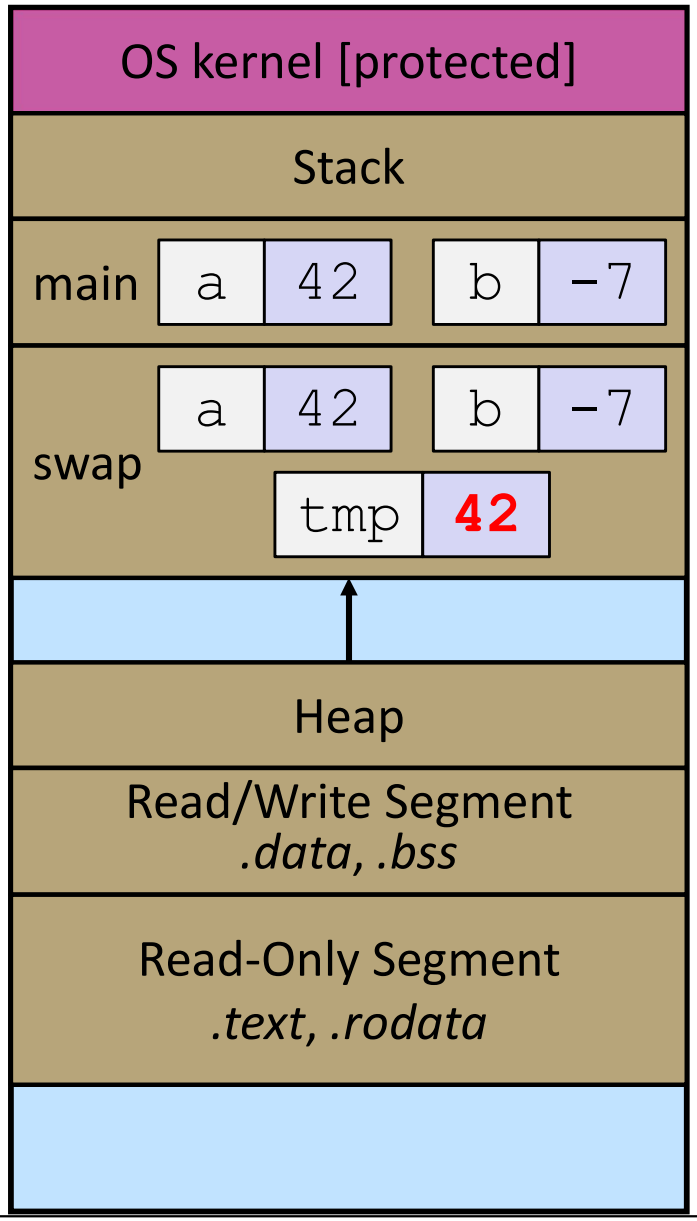
Broken Swap

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```



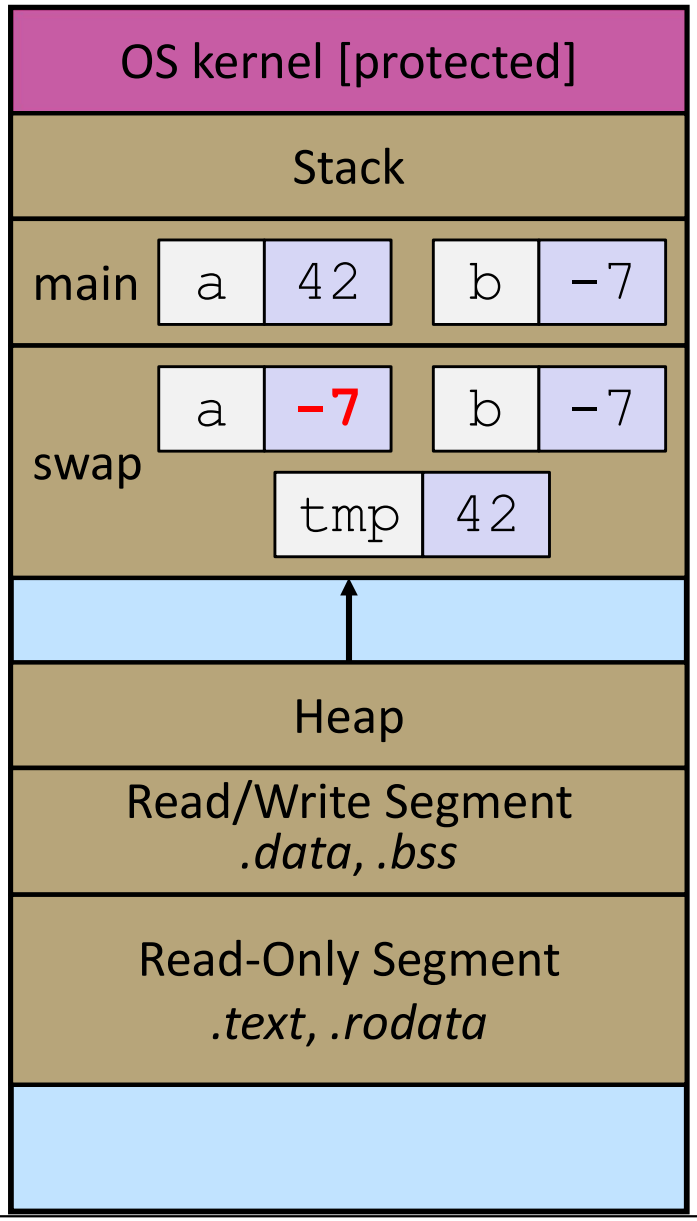
Broken Swap

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```



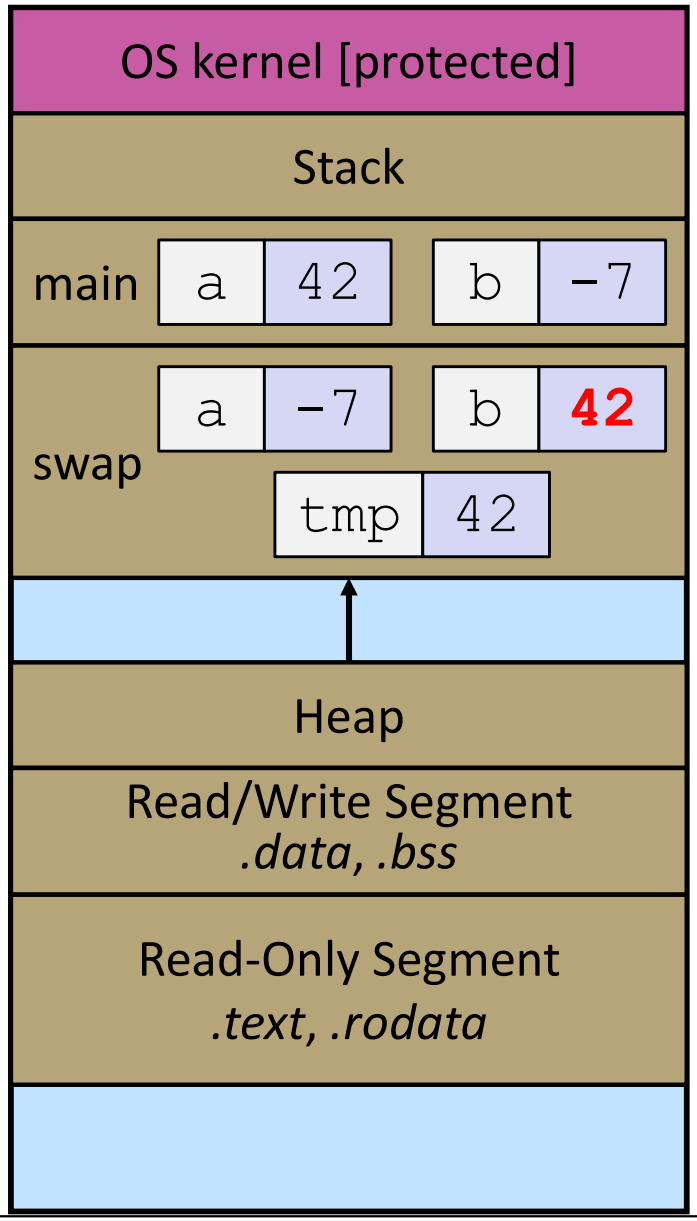
Broken Swap

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```



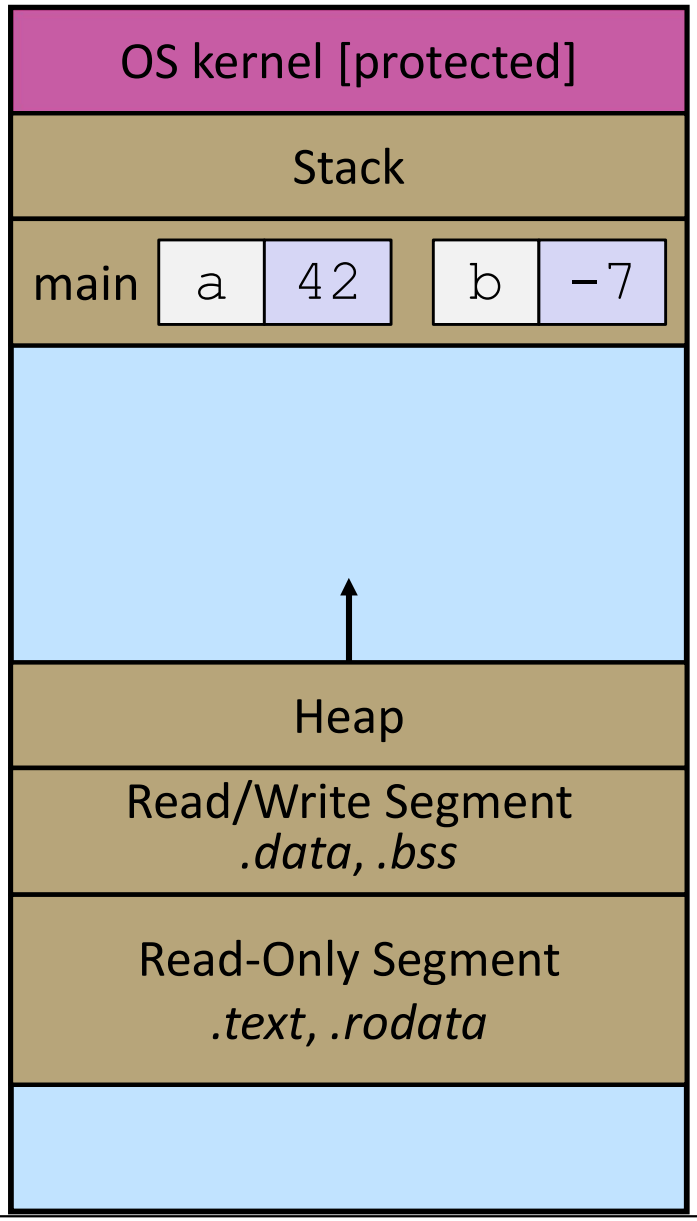
Broken Swap

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```



Broken Swap

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```



Let's Order Pizza

- But how many?
- Depends how many people we have
- Make a function to figure it out

Example : 1 Input Parameters, 1 output

```
int numLgPizzaToOrder (int numPeople)
{
        /* algorithm (formula) is from many years of experience.  */

        const int slicesPerPerson =3;
        int nPizzas;
        nPizzas = ceil(numPeople * slicesPerPerson / 8);
        return(nPizzas);
}
```

Memory usage by functions

2 styles: “Call-by-value” & "Call-by-Reference"

Variables	Address	Value
x	04902340	00000001
numStudents (in main)	04902348	00010110
	04902356	11011101
	04902364	01010000
i	04902372	00101100
	04902380	11011110
	04902388	01010000

“Call-by-**value**”:

- provide function with the value held in a variable input
- **Copies** the value to new internal variable

Variables	Address	Value
	04902340	00000001
numStudents (in main)	04902348	00010110
	04902356	11011101
	04902364	01010000
	04902372	00100110
numPeople (in numLgPizzaToOrder)	04902380	11011110
→	04902388	00010110

Copies the **Value** to an address used by the function.

“Call-by-value”

- ERASES the <u>local</u> value when leaving the function.
- Why? The variable is 'out of scope' & gone

Variables	Address	Value
	04902340	00000001
numStudents (in main)	04902348	00010110
	04902356	11011101
	04902364	01010000
	04902372	00100110
numPeople (in numLgPizzaToOrder)	04902380	11011110
	04902388	00010110

Copies the **<u>Value</u>** to an address used by the function.

“Call-by-reference”

- Does not copy
- Sends the ADDRESS of the original variable
- Allows you to change the value of the original variable

numStudents
(in main)

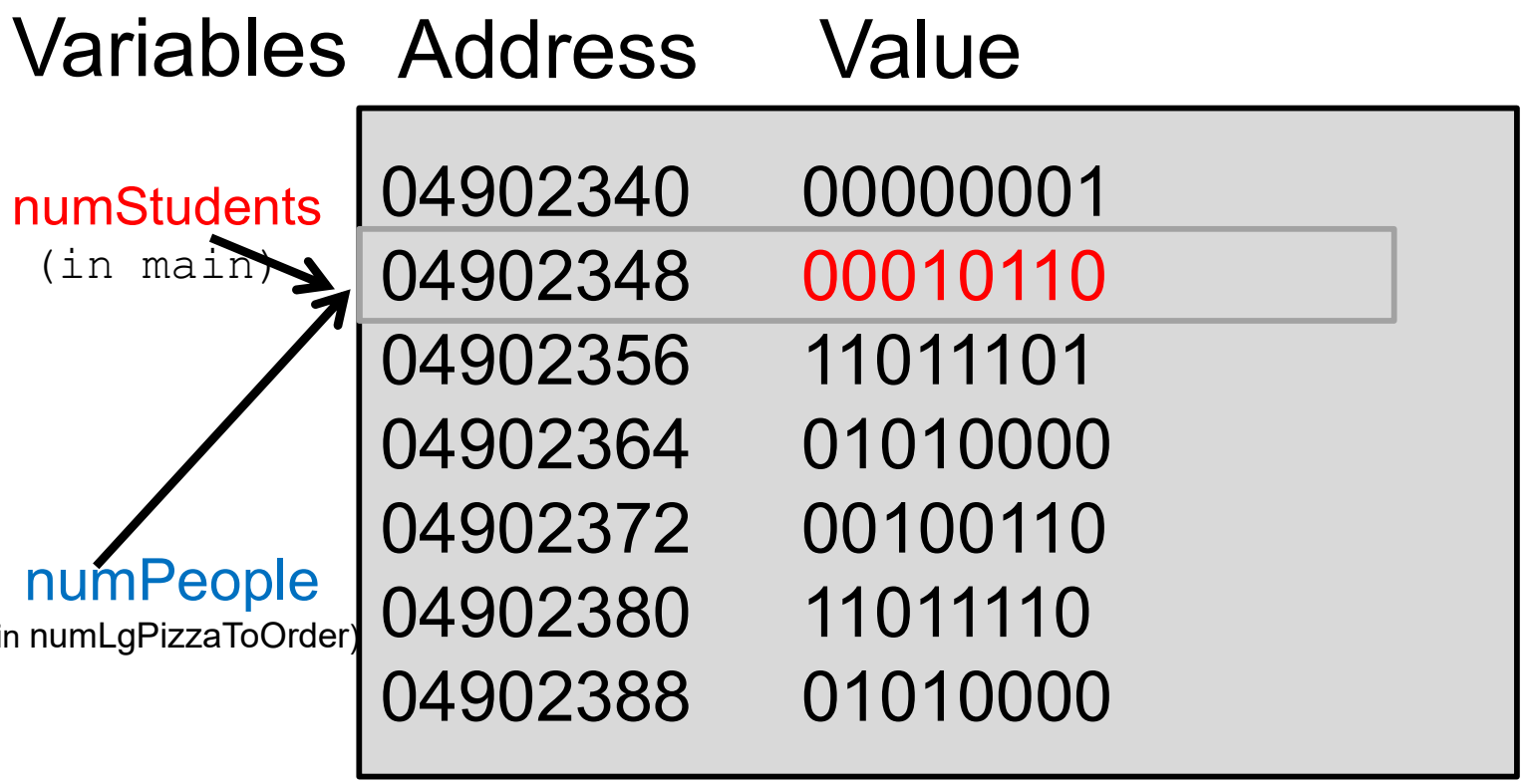
numPeople
(in numLgPizzaToOrder)

Variables	Address	Value
	04902340	00000001
	04902348	00010110
	04902356	11011101
	04902364	01010000
	04902372	00101100
	04902380	11011110
	04902388	01010000

Address Of

&

Notice that:
address of numStudents = address of numPeople below



Call-by-Reference Syntax

- Use & to indicate a variable is called by reference
- Use & both in declaration and definition
- & can be located anywhere between the type and variable

Call-by-Reference Example

Declaration (top of file). Matches definition,but add ;

```
void get_letters(char& letter1, char& letter2);
```

Definition (preferably above main)

```
void get_letters(char& letter1, char& letter2)
{
   cout << "Enter two letters: ";
   cin >> letter1 >> letter2;
}
```

Calling the function

```
main()
{
   char a,b;
   get_letters(a,b);
   cout << "After get_letters " ;
   cout << "a= " << a << " b= " << b<< endl;
}
```

Call-by-reference vs. Call-by-value

- Call-by-value preserves the value of the original input argument
- Call-by-reference can change the value of the original input argument
  - Effectively allows return of multiple values from function

```
int mysteryFunc(int& num1);

int main()
{
  int a=5;
  cout << mysteryFunc(a) << endl;
  cout << a << endl;
  return 0;
}

int mysteryFunc(int &num1)
{
  num1 += 3;
  return num1/4;
}
```

What does this do?

```
int mysteryFunc2(int inNum);

int main()
{
  int a=3;
  cout << mysteryFunc2(a);
  cout << a;
  return 0;
}

int mysteryFunc2(int inNum)
{
  inNum = inNum*inNum;
  return inNum;
}
```

What does this do?

Call-by-reference: Input arguments

- Arguments must be variables

  If declaration is:

```
void myFunc(float& inputNum);
```

```
Good call:
float inputVariable;
myFunc(inputVariable);
```

```
myFunc(25.4);
```
BAD call. Why?

A puzzle (and interview question)

- Write swap without using a temporary variable

A puzzle (clever)

```
swap(int &x, int & y)
{
        x = x + y;
        y = x - y; // (x+y) – y   (so orig x, what we want)
        x = x - y; // (x+y)  - y ( y is the orig x)
}
```

Code it now: More usage of &

```
int x = 5;
int& y=x; // y and x point to same address
y=10;

cout << x << endl; // output x value
cout << &x << endl; // output x address
cout << y << endl; // output y value
cout << &y << endl; // output y address
```

Overloading Functions

- Overloaded functions have the same name but different parameter lists
- Can be used to create functions that perform the same task but take different parameter types or different number of parameters
- Compiler will determine which version of function to call by argument and parameter lists

Function Overloading Examples

Using these overloaded functions,

```
void getDimensions(int);              // 1
void getDimensions(int, int);         // 2
void getDimensions(int, double);      // 3
void getDimensions(double, double);// 4
```

the compiler will use them as follows:

```
int length, width;
double base, height;
getDimensions(length);              // 1
getDimensions(length, width);       // 2
getDimensions(length, height);      // 3
getDimensions(height, base);        // 4
```

Program 6-27

```
// This program uses overloaded functions.
#include <iostream>
#include <iomanip>
using namespace std;

// Function prototypes
int square(int);
double square(double);

int main()
{
   int userInt;
   double userFloat;

   // Get an int and a double.
   cout << fixed << showpoint << setprecision(2);
   cout << "Enter an integer and a floating-point value: ";
   cin >> userInt >> userFloat;

   // Display their squares.
   cout << "Here are their squares: ";
   cout << square(userInt) << " and " << square(userFloat);
   return 0;
}
```

The overloaded functions have different parameter lists

Passing a double

Passing an int

(Program Continues)

Program 6-27 *(Continued)*

```
26  //*************************************************************
27  // Definition of overloaded function square.                 *
28  // This function uses an int parameter, number. It returns the *
29  // square of number as an int.                               *
30  //*************************************************************
31
32  int square(int number)
33  {
34     return number * number;
35  }
36
37  //*************************************************************
38  // Definition of overloaded function square.                 *
39  // This function uses a double parameter, number. It returns  *
40  // the square of number as a double.                         *
41  //*************************************************************
42
43  double square(double number)
44  {
45     return number * number;
46  }
```

Program Output with Example Input Shown in Bold

```
Enter an integer and a floating-point value: 12 4.2 [Enter]
Here are their squares: 144 and 17.64
```

Thank you