CSI33 Data Structures

Department of Mathematics and Computer Science
Bronx Community College

Outline

Declarations, Definitions and Prototypes

Function Declarations (Prototypes)

Any function can be declared more than once. A function declaration includes:

- The return type of the function (may be `void`).
- The name of the function.
- In parentheses, the formal parameters with their types. Default values may be provided.
- A semicolon to terminate the declaration.

```
int sum(int n1, int n2, int n3);
```

Declarations, Definitions and Prototypes

Function Definitions

A function definition consists of the:

- the function header. This looks like the function's declaration, except that there are no default values and there is no semicolon; and
- the function body, which consists of a code block (in braces). The code should use the formal parameters in the header. If the return type is not `void` there must be a return statement.

```
int sum(int n1, int n2, int n3) {
   return n1+n2+n3; }
```

Declarations, Definitions and Prototypes

Using Function Prototypes (Declarations)

C++ must see a function declaration or definition before it can translate a call to that function.
(Formal parameters cannot be allocated if their type is not known.)

Declarations, Definitions and Prototypes

A Function Prototype (Declaration) and Definition

```
double f(double x, double y);
int main()
{
   double a = 2.5, b = 3.0, c;
   c = f(a, b);
   return 0;
}
double f(double x, double y)
{
   return x * x + 2 * x * y;
```

Parameter Passing - Pass by Value

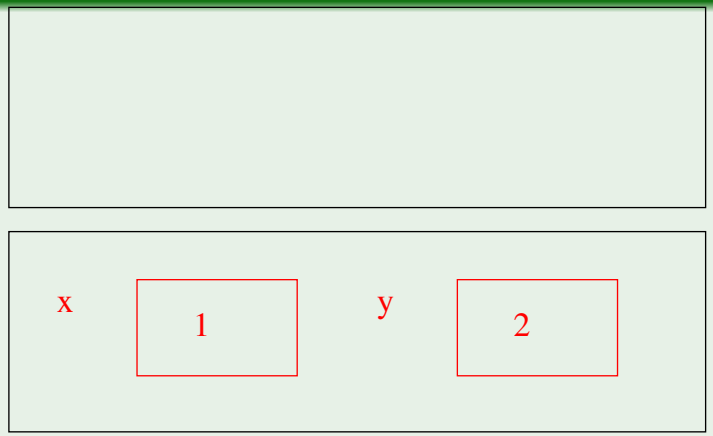
Pass by Value

- A separate copy of the actual parameter value is made, and assigned to the formal parameter.
- The code in the body of the function only affects the copy, not the original actual parameter.

Parameter Passing - Pass by Value

Pass by Value - Example

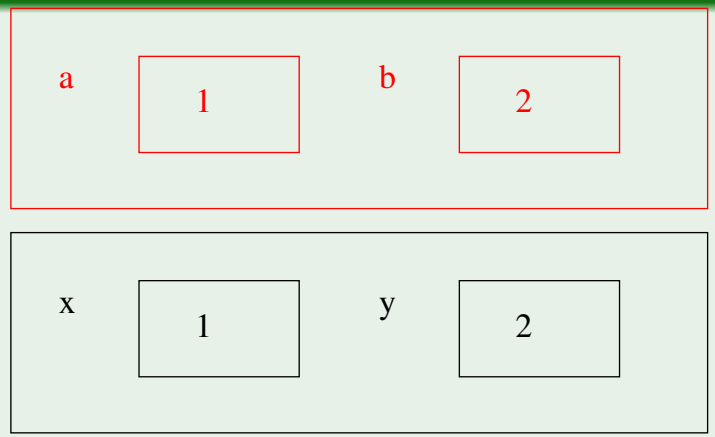
```
void f(int a, int b)
{
   cout << a << " " << b << endl;
   a += 3;
   b += 5;
   cout << a << " " << b << endl;
}
int main()
{
   int x = 1, y = 2;
   f(x, y);
   cout << x << " " << y << endl;
}
```



Parameter Passing - Pass by Value

Pass by Value - Example

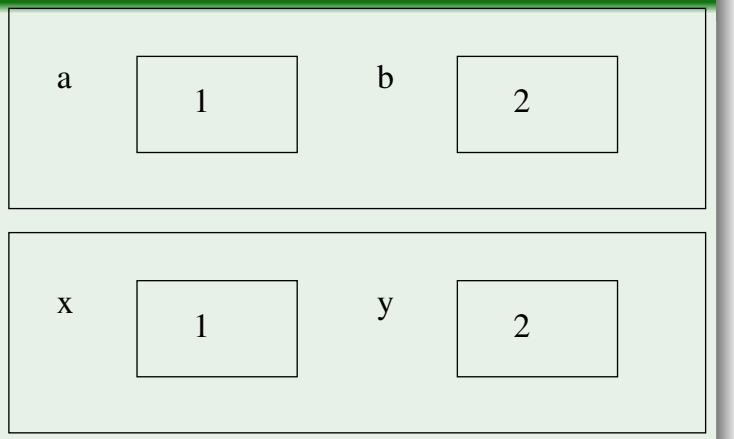
```
void f(int a, int b)
{
   cout << a << " " << b << endl;
   a += 3;
   b += 5;
   cout << a << " " << b << endl;
}
int main()
{
   int x = 1, y = 2;
   f(x, y);
   cout << x << " " << y << endl;
}
```



Parameter Passing - Pass by Value

Pass by Value - Example

```
void f(int a, int b)
{
   cout << a << " " << b << endl;
   a += 3;
   b += 5;
   cout << a << " " << b << endl;
}
int main()
{
   int x = 1, y = 2;
   f(x, y);
   cout << x << " " << y << endl;
}
```

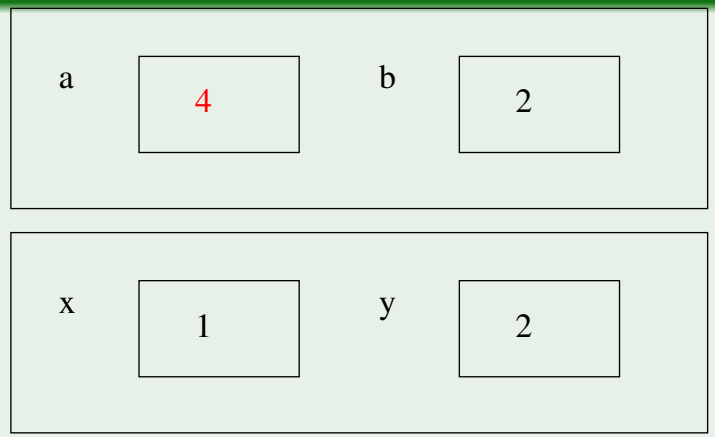


1 2

Parameter Passing - Pass by Value

Pass by Value - Example

```
void f(int a, int b)
{
   cout << a << " " << b << endl;
   a += 3;
   b += 5;
   cout << a << " " << b << endl;
}
int main()
{
   int x = 1, y = 2;
   f(x, y);
   cout << x << " " << y << endl;
}
```

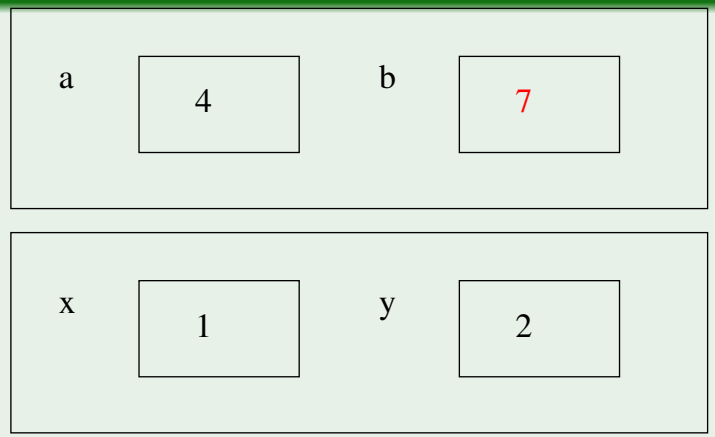



```
1 2
```

Parameter Passing - Pass by Value

Pass by Value - Example

```
void f(int a, int b)
{
   cout << a << " " << b << endl;
   a += 3;
   b += 5;
   cout << a << " " << b << endl;
}
int main()
{
   int x = 1, y = 2;
   f(x, y);
   cout << x << " " << y << endl;
}
```

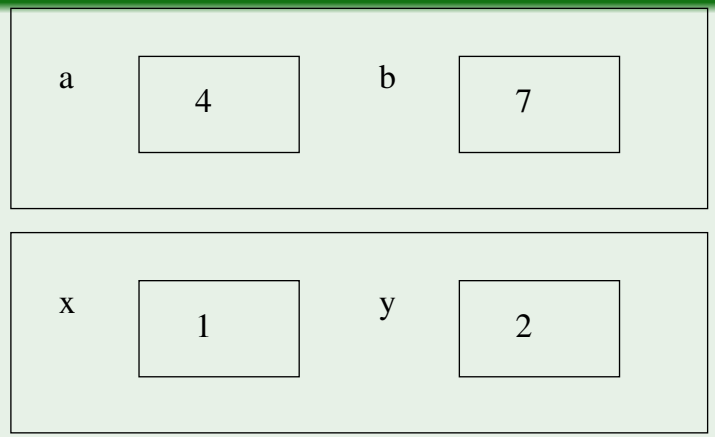


1 2

Parameter Passing - Pass by Value

Pass by Value - Example

```
void f(int a, int b)
{
   cout << a << " " << b << endl;
   a += 3;
   b += 5;
   cout << a << " " << b << endl;
}
int main()
{
   int x = 1, y = 2;
   f(x, y);
   cout << x << " " << y << endl;
}
```

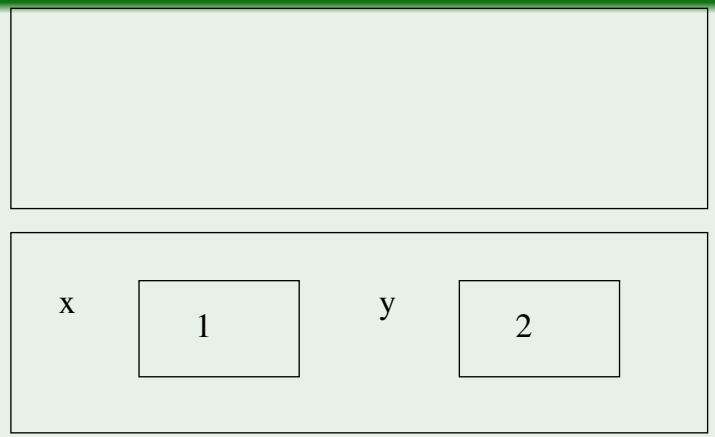



```
1 2
4 7
```

Parameter Passing - Pass by Value

Pass by Value - Example

```
void f(int a, int b)
{
   cout << a << " " << b << endl;
   a += 3;
   b += 5;
   cout << a << " " << b << endl;
}
int main()
{
   int x = 1, y = 2;
   f(x, y);
   cout << x << " " << y << endl;
}
```




```
1 2
4 7
1 2
```

Parameter Passing - Pass by Reference

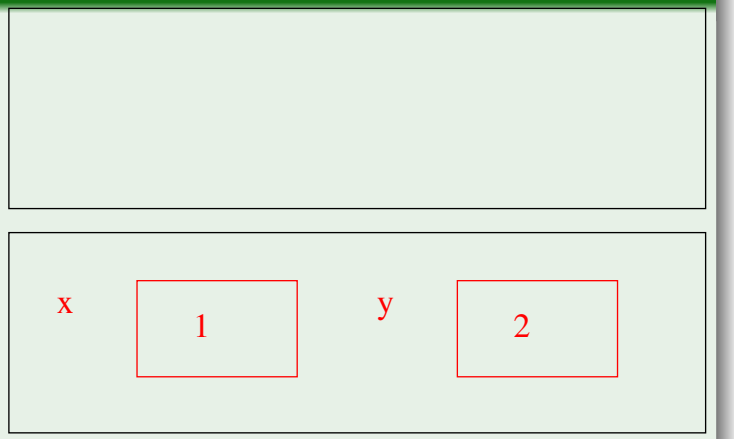
Pass by Reference

- An ampersand (&) before the formal parameter name in the header indicates passing by reference.
- A parameter passed by reference must be a variable.
- A reference to its value (its address) is passed to the called function.
- The function can use or change this value.
- If the value is changed, the actual parameter passed by reference has the changed value even after the function call is completed.

Parameter Passing - Pass by Reference

Pass by Reference Example

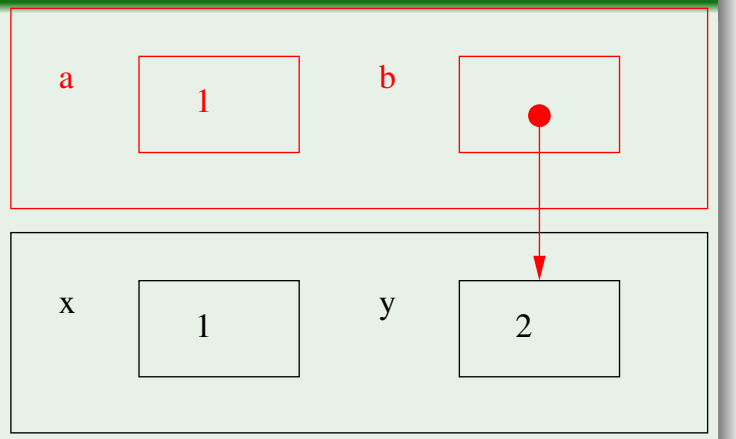
```
void f(int a, int &b)
{
   cout << a << " " << b << endl;
   a += 3;
   b += 5;
   cout << a << " " << b << endl;
}
int main()
{
   int x = 1, y = 2;
   f(x, y);
   cout << x << " " << y << endl;
}
```



Parameter Passing - Pass by Reference

Pass by Reference Example

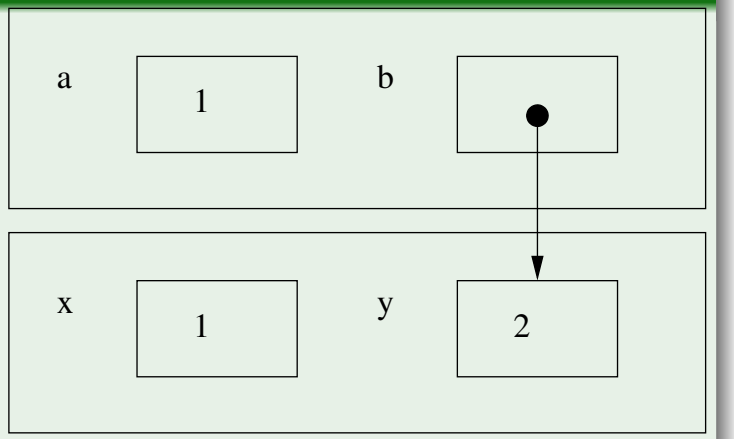
```
void f(int a, int &b)
{
   cout << a << " " << b << endl;
   a += 3;
   b += 5;
   cout << a << " " << b << endl;
}
int main()
{
   int x = 1, y = 2;
   f(x, y);
   cout << x << " " << y << endl;
}
```



Parameter Passing - Pass by Reference

Pass by Reference - Example

```
void f(int a, int &b)
{
   cout << a << " " << b << endl;
   a += 3;
   b += 5;
   cout << a << " " << b << endl;
}
int main()
{
   int x = 1, y = 2;
   f(x, y);
   cout << x << " " << y << endl;
}
```

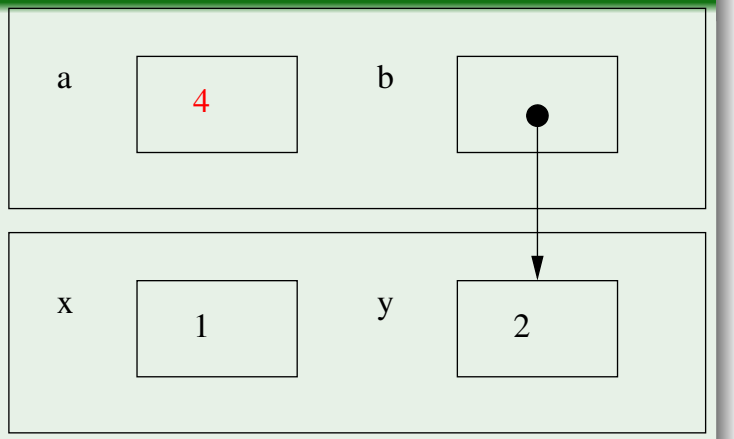


1 2

Parameter Passing - Pass by Reference

Pass by Reference - Example

```
void f(int a, int &b)
{
   cout << a << " " << b << endl;
   a += 3;
   b += 5;
   cout << a << " " << b << endl;
}
int main()
{
   int x = 1, y = 2;
   f(x, y);
   cout << x << " " << y << endl;
}
```

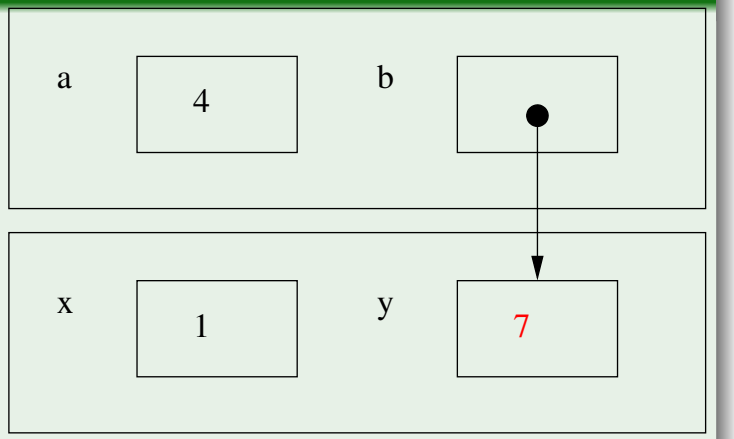


1 2

Parameter Passing - Pass by Reference

Pass by Reference - Example

```
void f(int a, int &b)
{
   cout << a << " " << b << endl;
   a += 3;
   b += 5;
   cout << a << " " << b << endl;
}
int main()
{
   int x = 1, y = 2;
   f(x, y);
   cout << x << " " << y << endl;
}
```

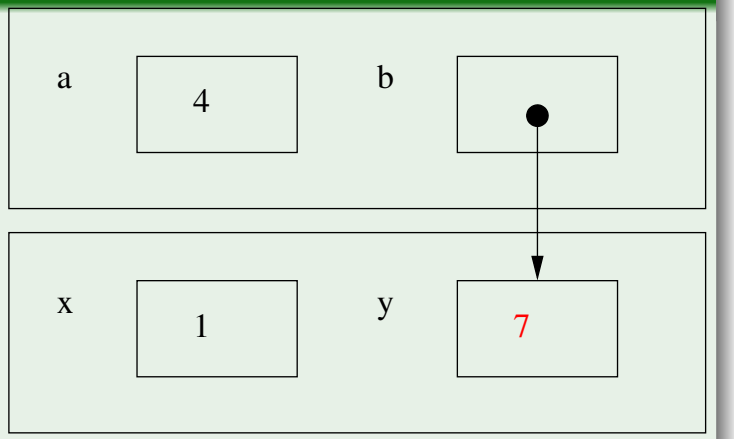


1 2

Parameter Passing - Pass by Reference

Pass by Reference - Example

```
void f(int a, int &b)
{
   cout << a << " " << b << endl;
   a += 3;
   b += 5;
   cout << a << " " << b << endl;
}
int main()
{
   int x = 1, y = 2;
   f(x, y);
   cout << x << " " << y << endl;
}
```



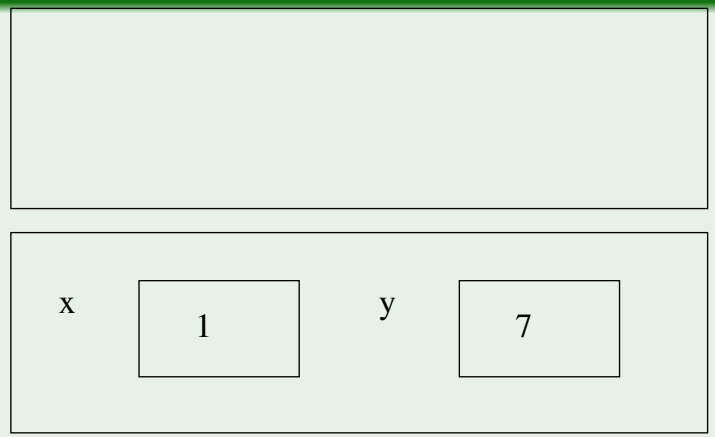
1 2

4 7

Parameter Passing - Pass by Reference

Pass by Reference - Example

```
void f(int a, int &b)
{
   cout << a << " " << b << endl;
   a += 3;
   b += 5;
   cout << a << " " << b << endl;
}
int main()
{
   int x = 1, y = 2;
   f(x, y);
   cout << x << " " << y << endl;
}
```




```
1 2
4 7
1 7
```

Passing Arrays as Parameters

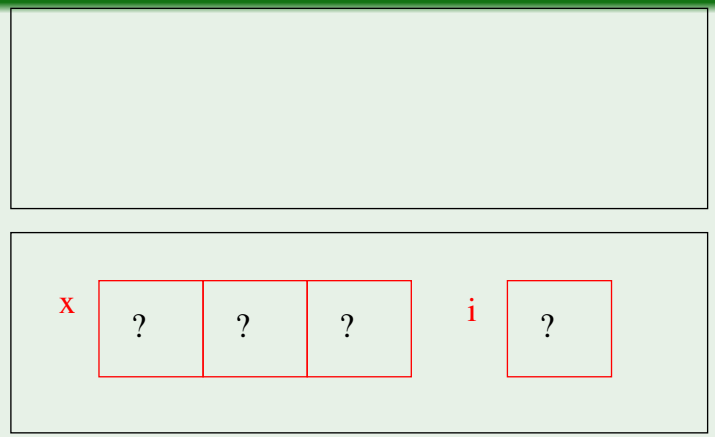
Passing Arrays as Parameters

- Arrays are automatically passed by reference (no ampersand is used).
- In the function header, the name of any array parameter is followed by empty brackets (`[]`).
- The function does not need to know the length of the array, but it is a good idea to give it as a separate `int` parameter.

Passing Arrays as Parameters

Passing Arrays as Parameters - Example

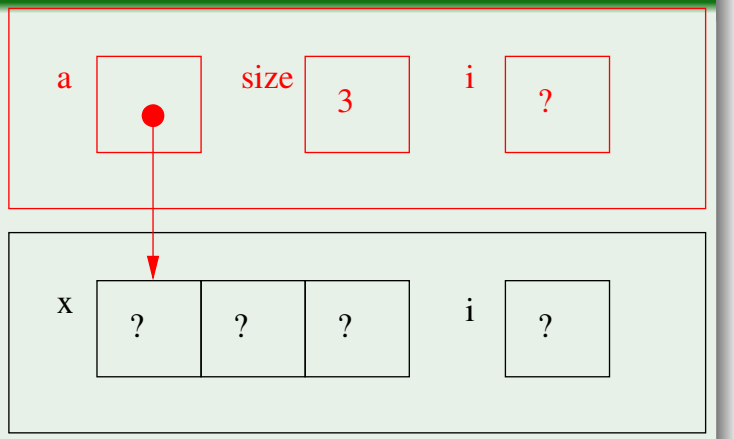
```
void clear(int a[], int size)
{
   int i;
   for (i=0; i < size; i++)
      a[i] = 0;
}
int main()
{
   int x[3], i;
   clear(x, 3);
   for (i=0; i < size; i++)
      cout << x[i] << ‘‘ ’’;
   return 0;
```



Passing Arrays as Parameters

Passing Arrays as Parameters - Example

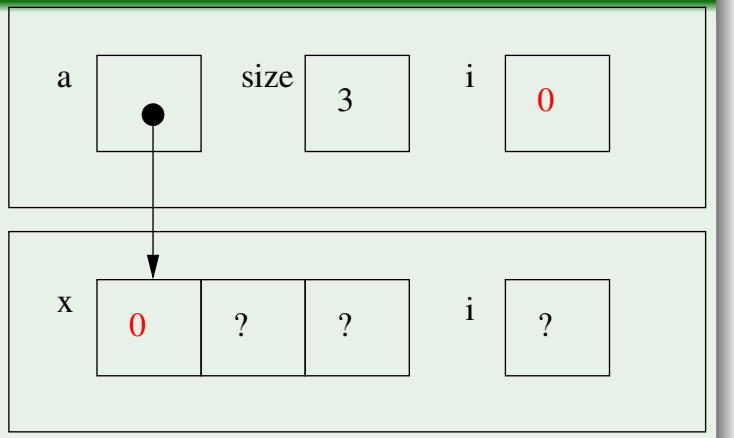
```
void clear(int a[], int size)
{
   int i;
   for (i=0; i < size; i++)
      a[i] = 0;
}
int main()
{
   int x[3], i;
   clear(x, 3);
   for (i=0; i < size; i++)
      cout << x[i] << ‘‘ ’’;
   return 0;
```



Passing Arrays as Parameters

Passing Arrays as Parameters - Example

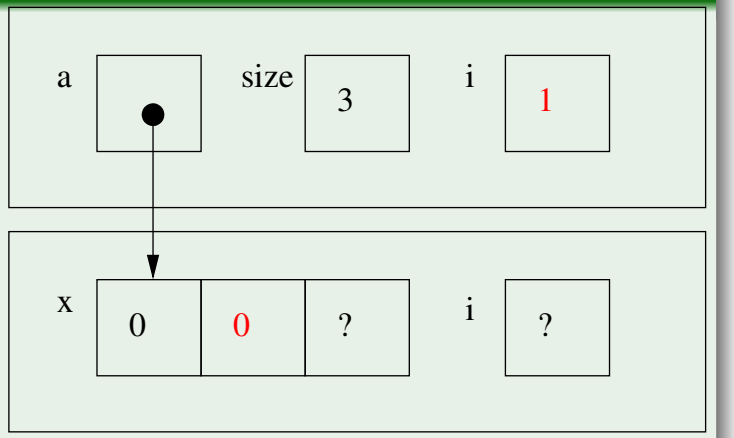
```
void clear(int a[], int size)
{
   int i;
   for (i=0; i < size; i++)
      a[i] = 0;
}
int main()
{
   int x[3], i;
   clear(x, 3);
   for (i=0; i < size; i++)
      cout << x[i] << ‘‘ ’’;
   return 0;
```



Passing Arrays as Parameters

Passing Arrays as Parameters - Example

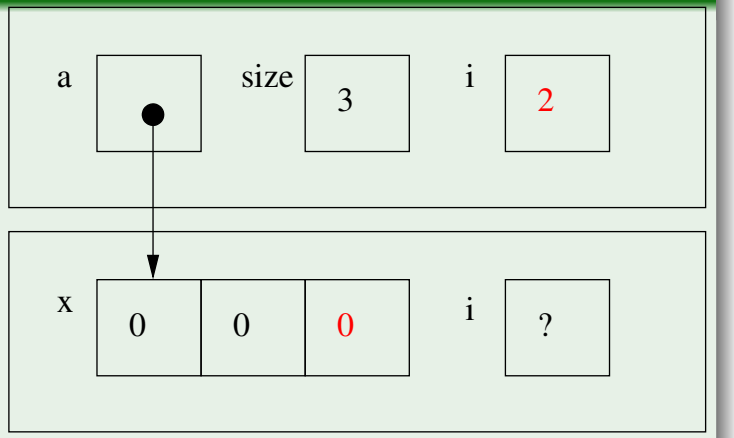
```
void clear(int a[], int size)
{
   int i;
   for (i=0; i < size; i++)
      a[i] = 0;
}
int main()
{
   int x[3], i;
   clear(x, 3);
   for (i=0; i < size; i++)
      cout << x[i] << ‘‘ ’’;
   return 0;
```



Passing Arrays as Parameters

Passing Arrays as Parameters - Example

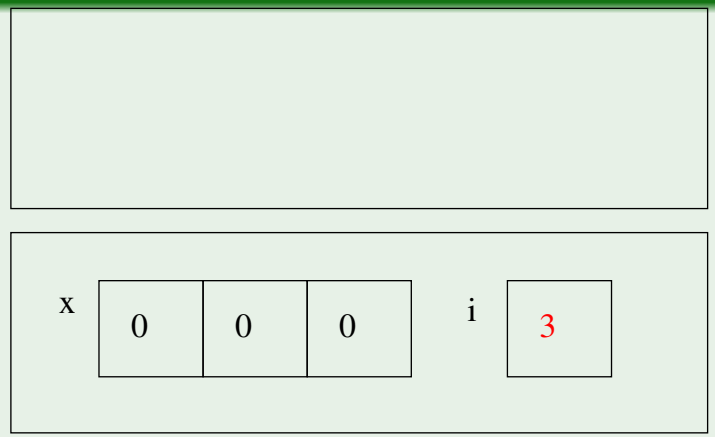
```
void clear(int a[], int size)
{
   int i;
   for (i=0; i < size; i++)
      a[i] = 0;
}
int main()
{
   int x[3], i;
   clear(x, 3);
   for (i=0; i < size; i++)
      cout << x[i] << ‘‘ ’’;
   return 0;
```



Passing Arrays as Parameters

Passing Arrays as Parameters - Example

```
void clear(int a[], int size)
{
   int i;
   for (i=0; i < size; i++)
      a[i] = 0;
}
int main()
{
   int x[3], i;
   clear(x, 3);
   for (i=0; i < size; i++)
      cout << x[i] << ‘‘ ’’;
   return 0;
```



0 0 0

const Parameters

const Parameters

Parameters whose values are not supposed to change are declared as `const` parameters.

This way, the compiler will give an error message if it finds any code which changes the value of that parameter in the body of the function.

const Parameters - Example

```
void f(const int a, int b)
{
   a = 2; // this will generate a compiler error
   b = 2; // this is fine
```

const Parameters

const Parameters

Parameters whose values are not supposed to change are declared as `const` parameters.

This way, the compiler will give an error message if it finds any code which changes the value of that parameter in the body of the function.

const Parameters - Example

```
void f(const int a, int b)
{
   a = 2; // this will generate a compiler error
   b = 2; // this is fine
```

Default Parameters

Default Parameters

Default parameters in C++ are similar to those in Python. That is, if a parameter value is left unspecified in a function call, the default value for that parameter will be automatically assigned.

Parameters which can get default values must be the last parameters in the parameter list.

Default Parameters

Default Parameters - Example

```
void f(int a, int b, int c=2, int d=3)
{
   // do something with the parameters
}
int main()
{
   f(0, 1); // equivalent to f(0, 1, 2, 3);
   f(4, 5, 6); // equivalent to f(4, 5, 6, 3);
   f(4, 5, 6, 7); // no default value used
   return 0;
}
```

Return Value

Return Value

Functions in C++ can return only one value.

This is not a significant limitation, it can be resolved by encapsulating multiple values in a single class and returning an instance of that class or by using *pass by reference*

Header Files

Header Files (Ext .h) Can Be # included Multiple Times

Different C++ files (extension .cpp) will often use the same library functions or member functions of the same class. Instead of re-declaring them (possibly incorrectly) use the preprocessor's `#include` directive to reuse the same file.

The usual benefits of code reuse apply: fewer errors, and easier maintenance, since a change of a declaration need only be made once.

So far we included the `iostream` and `string` header files.

Header Files

Multiple #includes Must Be Managed

#included files can #include other files, and so on, so there is a danger that some header file may be included multiple times in a single file by the preprocessor.

If this leads to multiple declarations of the same variable or class, it will cause a build error.

Header Files

#IFNDEF...#ENDIF

To prevent multiple inclusions of the file `conversions.h`, place the directives

```
#ifndef __CONVERSIONS_H
#define __CONVERSIONS_H
```

at the beginning of the header file, and

```
#endif
```

at the end.

These are pre-processor commands (start with # sign)

See programs conversions.cpp, conversions.h

Inline Functions

Include Files Can Have Function Definitions As Well As Declarations

If a function definition is small (say, one line of code) it can be placed in a header file for inclusion wherever it is needed.

The `inline` keyword prevents a function definition from being included twice, and suggests to the compiler that any function call to this function be replaced with the **actual code in the body of the function**.

This produces a faster program, since the overhead of a function call (copying parameter values, pushing onto and popping from the call stack) is removed.

A general rule: declare functions that are <= 5 lines long as `inline`

Inline Functions

Inline Functions - Example

```
// conversions2.h
#ifndef __CONVERSIONS_H
#define __CONVERSIONS_H

inline double f_to_c(double f=0.0)
{
   return (f - 32.0) * (5.0 / 9.0);
}
inline double c_to_f(double c=0.0)
{
   return (9.0 / 5.0) * c + 32.0;
}
#endif
```

Assert Statements

In C++, assert Has The Same Behavior as in Python

Like Python, C++ has an `assert` command that halts a program immediately if the given boolean expression is false.

It has the syntax `assert(<boolean expression>)`.
To be used, the C++ source file must contain the command
`#include <cassert>`

Assert Statements

Testing

Besides ensuring that preconditions are true, the assert statement can be used to check the truth of a postcondition, say, after a member function has been called.

The Scope and Lifetime of Variables

Scope

- The scope of a variable is the part of the program code in which it can be accessed.
- In C++, this is the code block in which the variable is declared.

The Scope and Lifetime of Variables

Lifetime

The lifetime of a variable is the period of time in the execution of the program code starting with the moment when the memory for the variable is allocated and ending when it is deallocated (i.e. in which the variable name is defined and has a value and meaning).

See the program scope.cpp

Common C++ Mistakes by Python Programmers

- forgetting the semicolon after each statement
- putting a semicolon after `for`, `if` or `while`
- putting a colon after `for`, `if` or `while`
- forgetting curly braces to mark code blocks
- forgetting parentheses around boolean expressions in `if` or `while`
- putting semicolon after heading in a function definition
- assigning one array variable to another (you must assign each entry individually by its index)

Sorting algorithms use

Selection sort and merge sort

Recall Selection sort and merge sort. Let's take a look at their C++ code: selection.cpp, sort.cpp, sort.h

Switch Statements in C++

Switch Statements in C++

C++ supports another decision statement that Python doesn't have: switch statement.

Note that any statement written with a switch statement can be written using `if` and `else` statements, but not vice versa.

Switch Statements in C++

Syntax of switch statement

```
switch(<var>) {
case <expr1>:
   code here
...
case <exprN>:
   code here
default:
   code here
}
```

expr1, ... exprN must be an *ordinal* value (i.e. its type must be int, char, or bool)

Switch Statements in C++

```
// switchWithAssert.cpp
#include <iostream>
#include <cassert>

using namespace std;
int main() {
   int choice;
   cout << "Enter your choice of 1, 2 or 3:";
   cin >> choice;

   assert (choice == 1 || choice == 2 || choice == 3);
```

Switch Statements in C++

```
    switch(choice) {
    case 1:
       cout << "You chose 1 \n";
    case 2:
       cout << "You chose 2 \n";
    case 3:
       cout << "You chose 3 \n";
    default:
       cout << "You made invalid choice.  \n";
    }
}
```

See more programs: switch.cpp, switch2.cpp, switch3.cpp

In-class work

Average functions

see the handout or the CSI33_Day17_InClassWork.pdf