Functions

CS 201

This slide set covers the basics of C++ functions. Please read Chapter 6 (6.1 - 6.17) from your Deitel & Deitel book.

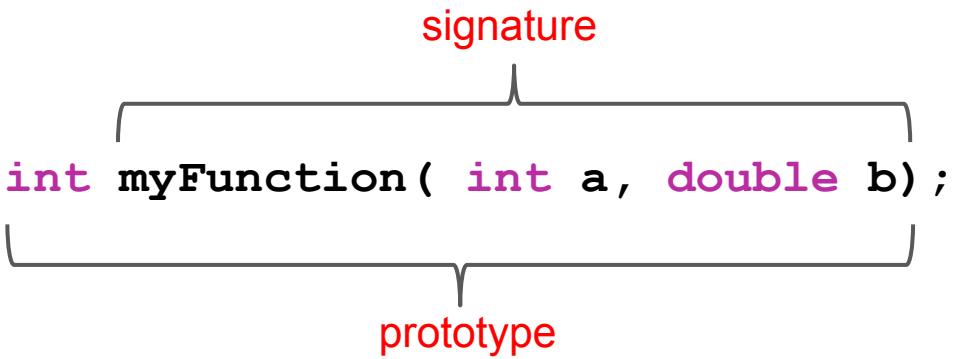
Introduction

- Divide and conquer technique
  - Construct a large program from small, simple pieces (e.g., components)
- Functions
  - Facilitate the design, implementation, operation, and maintenance of large programs
- C++ supports two types of functions
  - Member functions
  - Global functions

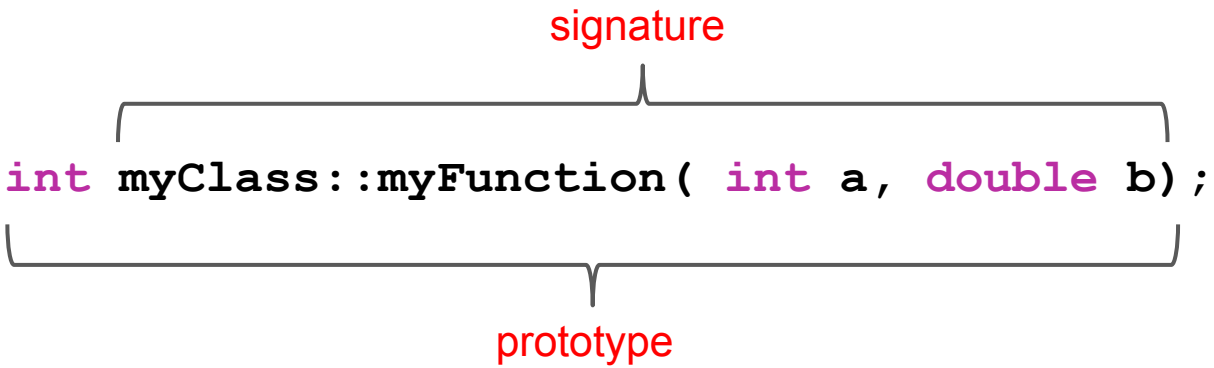
Global functions

- Do not belong to a particular class
- In C++, **`main`** is a global function
- Many functions in C++ Standard Library header files are global
  - For example, **`sqrt`** in the **`<cmath>`** header file
    - Call it as **`sqrt( 900.0 )`**
- You can also write your own global functions
  (although it is not an object-oriented programming)

Functions



*You can **overload** functions by using the same function name with different signatures.*



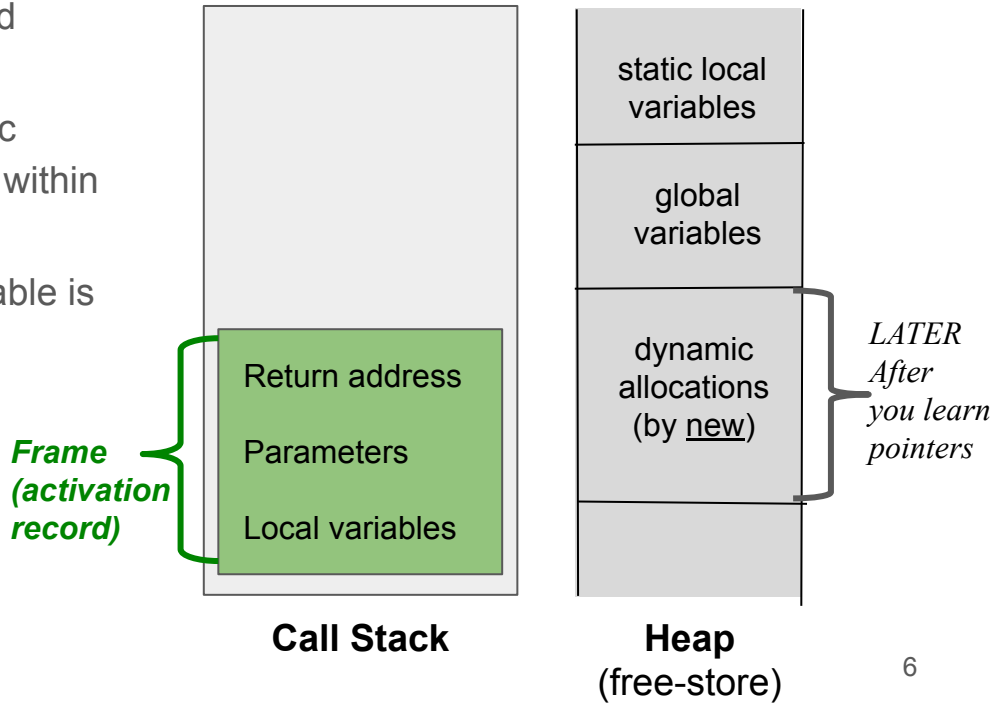
Example:

Write a global function which takes two int parameters and returns their largest

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

int findMax( int, int );

int main() {
   int larger = findMax( 42, 16 );
   cout << "The larger of 42 and 16 is " << larger << endl;
   return 0;
}

int findMax( int x, int y ) {
   if ( x > y )
      return x;
   else
      return y;
}
```

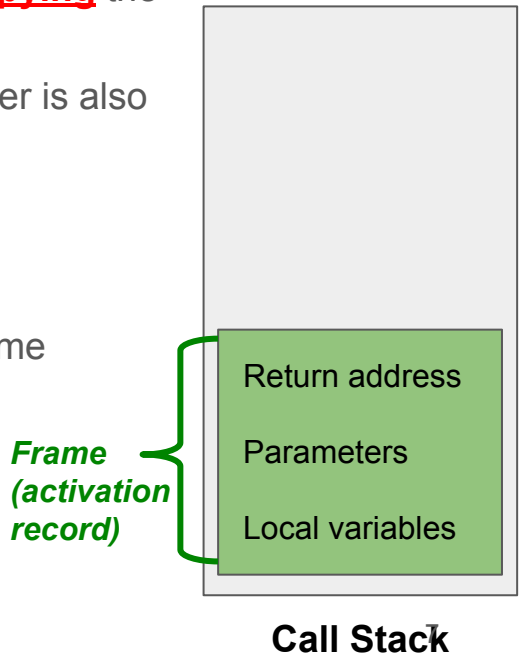
Function call stack

- Also called the program execution stack or run-time stack
- Call stack supports the **function call / return** mechanism
  - Functions use allocated storage called *frame* on the call stack
  - Function parameters and its non-static (automatic) local variables are stored within the function's frame
  - Storage size of a parameter or a variable is determined by its data type



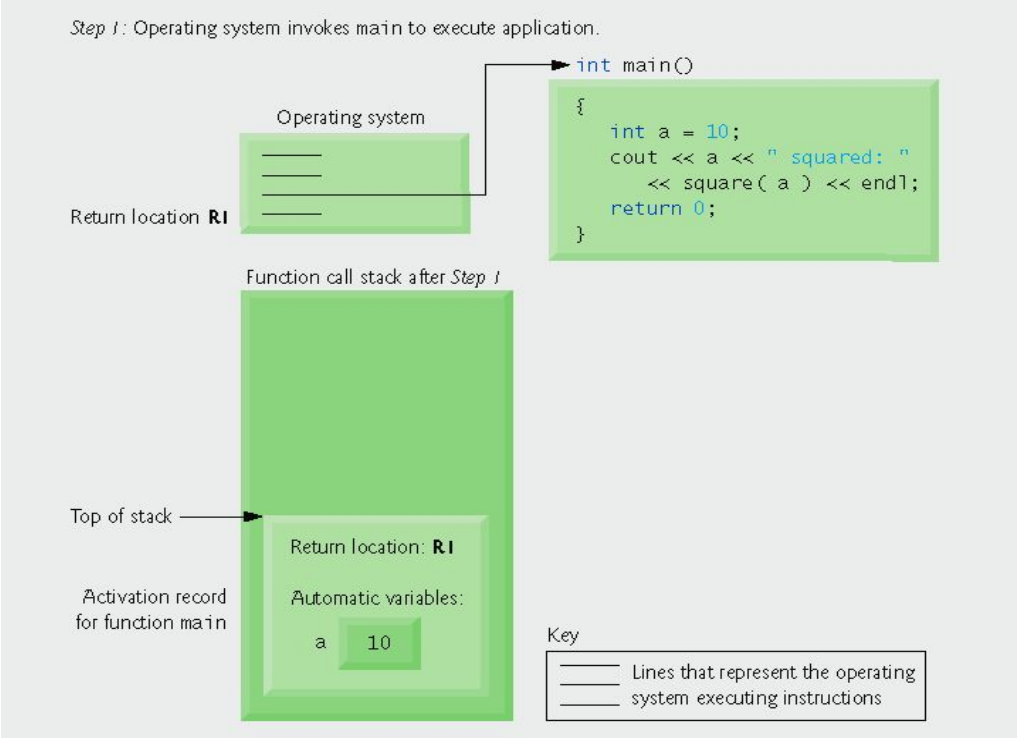
Function call stack

- Each time a function makes a call to another function
  - A new stack ***frame*** (also known as an ***activation record***) for the new function call is created and pushed onto the top of the stack
  - Default argument passing initializes the function parameters by **copying** the value of the function call arguments (known as pass-by-value)
  - Return address that the called function needs to return to in its caller is also maintained
- Each time a function returns
  - Stack frame for this function call is popped out from the stack
  - Control is transferred to the return address in the popped stack frame
- Stack overflow
  - Error that occurs when there are too many function calls such that the necessary space for their frames exceeds the size of the call stack (due to memory limitations)



How is this program executed? (Example for call stack & frames)

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

int square( int ); //prototype

int main() {
    int a = 10;
    cout << a << " squared: "
         << square( a ) << endl;
    return 0;
}

// returns the square of an integer
int square( int x ) {
    return x * x;
}
```



Output

```
10 squared: 100
```

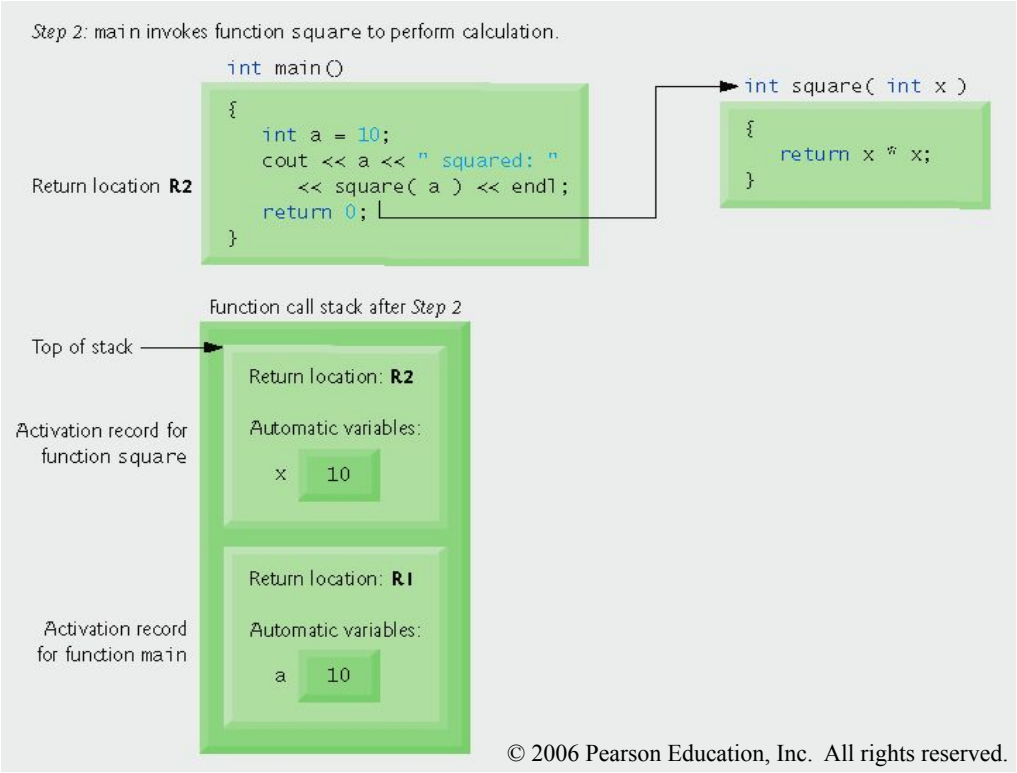
Function call stack ***after the operating system invokes*** **`main`** to execute the application

How is this program executed? (Example for call stack & frames)

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

int square( int ); //prototype

int main() {
    int a = 10;
    cout << a << " squared: "
         << square( a ) << endl;
    return 0;
}

// returns the square of an integer
int square( int x ) {
    return x * x;
}
```

Output

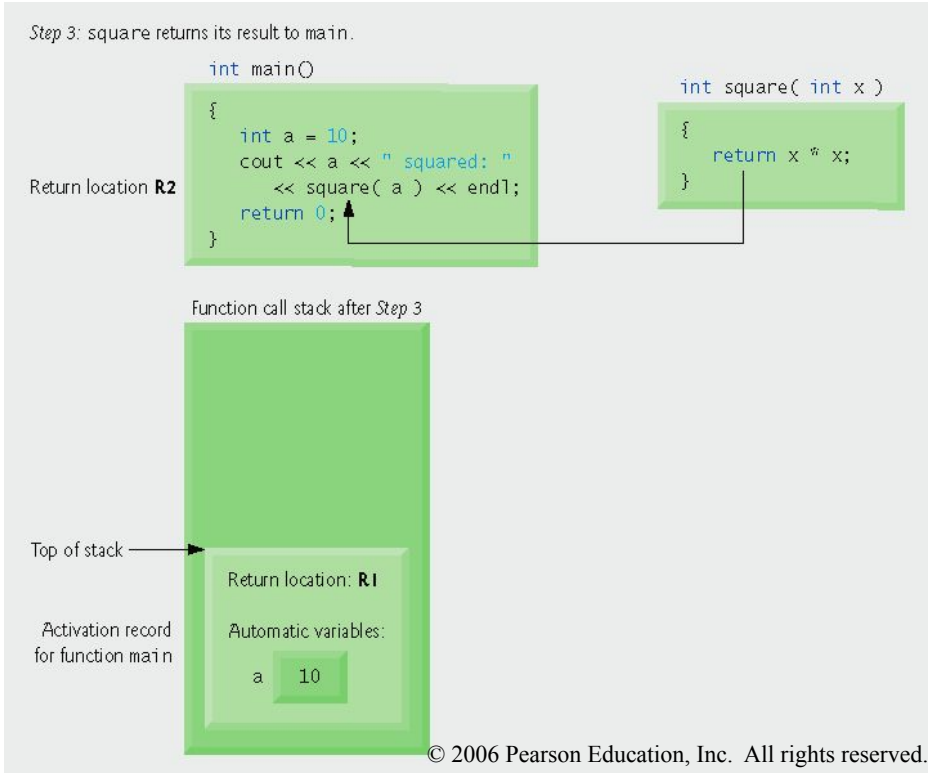
```
10 squared: 100
```



Function call stack ***after*** **`main`** ***invokes the function*** **`square`** to perform the calculation

How is this program executed? (Example for call stack & frames)

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

int square( int ); //prototype

int main() {
    int a = 10;
    cout << a << " squared: "
         << square( a ) << endl;
    return 0;
}

// returns the square of an integer
int square( int x ) {
    return x * x;
}
```



Output

```
10 squared: 100
```

Function call stack ***after the function*** **`square`** ***returns to*** **`main`**

Parameter passing in C++

- Two ways to pass arguments to functions
- **Pass-by-value**
  - **Copy** of the argument's value is passed to the called function
  - Changes to the copy do not affect the original argument's value in the caller
    - Prevents accidental side effects of functions
  - Default parameter passing
- **Pass-by-reference**
  - Gives called function the ability to access and modify the caller's argument data directly

Pass-by-reference

- **Reference variable**
  - An alias, which is another name for an already existing variable
  - **&** is placed after the variable type
  - A reference should be initialized with a variable. Then, this variable may be referenced by either the variable name or the reference name

```
int count;
int& no = count;
count = 4;
cout << no << endl;
```

- **Reference parameter**
  - An alias for its corresponding argument in a function call
  - **&** is placed after the parameter type in the function prototype and the function header
  - Data types of the argument and the reference parameter SHOULD be the same
  - Parameter name in the body of the called function actually refers to the original variable in the calling function

```
void bar( int& id ) {
  id = 10;
}
void foo ( ) {
  int a = 5;
  bar( a );
  cout << a << endl;
}
```

Written as "id is a reference to an int" (from right to left)

Pass-by-value and pass-by-reference for primitive types

```
#include <iostream>
using namespace std;
void nextIntByValue( int );
int main() {
    int x = 10;
    nextIntByValue( x );
    cout << x << endl;
    return 0;
}
void nextIntByValue( int y ) {
    y++;
}
```

`nextIntByValue` uses **pass-by-value**

What is the output of this program?

```
#include <iostream>
using namespace std;
void nextIntByReference( int& );
int main() {
    int x = 20;
    nextIntByReference( x );
    cout  << x << endl;
    return 0;
}
void nextIntByReference( int& y ) {
    y++;
}
```

`nextIntByReference` uses **pass-by-reference**

What is the output of this program?

Pass-by-value and pass-by-reference for objects

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

class Student {
public:
   Student( int i = 0 );
   void setId( int i );
   int getId();
private:
   int id;
};
Student::Student( int i ) {
   setId(i);
}
void Student::setId( int i ) {
   id = i;
}
int Student::getId( ) {
   return id;
}
```

```
void nextStudentIdByValue( Student s ) {
     int x = s.getId();
     x++;
     s.setId( x );
}
int main() {
     Student s1( 20 );
     nextStudentIdByValue( s1 );
     cout << s1.getId() << endl;
     return 0;
}
```

`nextStudentIdByValue` uses **pass-by-value**

What is the output of this program?

BE CAREFUL: That is different than Java!!!

Pass-by-value and pass-by-reference for objects

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

class Student {
public:
   Student( int i = 0 );
   void setId( int i );
   int getId();
private:
   int id;
};
Student::Student( int i ) {
   setId(i);
}
void Student::setId( int i ) {
   id = i;
}
int Student::getId( ) {
   return id;
}
```

```
void nextStudentIdByReference( Student& s ) {
    int x = s.getId();
    x++;
    s.setId( x );
}
int main() {
    Student s2( 40 );
    nextStudentIdByReference( s2 );
    cout << s2.getId() << endl;
    return 0;
}
```

`nextStudentIdByReference` uses **pass-by-reference**

What is the output of this program?

Remarks on parameter passing in C++

- Disadvantage of pass-by-value
  - If a large data item is passed, copying that data can take a considerable amount of execution time and memory space

- Advantages of pass-by-reference
  - When used for objects, it is good for performance reasons, because it eliminates the pass-by-value overhead of copying large amounts of data
  - A useful way to be able to return more than one value

- Disadvantages of pass-by-reference
  - Pass-by-reference can weaken security since the called function can corrupt the caller's data
  - Hard to tell from the function call if it is pass-by-reference (*so cannot tell if it will be modified*)
  - Cannot pass a literal value (or an expression or a variable of a different but compatible type) to a reference type parameter (e.g., `nextIntByRef( 4 )` is an invalid call)

Principle of least privilege

- *Functions should not be given the capability to modify its parameters unless it is absolutely necessary*

- **`const` qualifier** is used to enforce this principle when defining parameters
  - `void example( const int x, const GradeBook& G);`
    - `const int x` // parameter x cannot be modified in the function
    - `const GradeBook& G` // parameter G is a constant reference to an object, so it cannot be modified in the function and it cannot call a non-const member function

- Using the principle of least privilege to properly design software
  - can greatly reduce debugging time
  - can greatly reduce improper side effects
  - can make a program easier to modify and maintain

Principle of least privilege

```
#include <iostream>
using namespace std;
class Book{
public:
   void setNo( int i );
   int getNo( ) const;
   void displayNo( );
   int publicNo;
private:
   int no;
};
void Book::setNo( int n ){
    no = n;
}
int Book::getNo( ) const{
    return no;
}
void Book::displayNo( ){
    cout << no << endl;
}
```

```
void foo( const Book& B ){
   // B is a constant reference, thus it CANNOT change
   // a data member or call a non-const member function

   // THREE STATEMENTS BELOW LEAD TO COMPILE-TIME ERRORS
   // B.publicNo = 10;
   // B.setNo(20);
   // B.displayNo();
   cout << B.getNo() << endl;
}
int main() {
    Book A;
    foo( A );
    return 0;
}
```

For passing large objects, use a constant reference parameter to simulate the appearance and security of pass-by-value and avoid the overhead of passing a copy of the large object

Exercise (returning more than one value):

Write a global function which takes two int parameters and computes and returns the sum of the inputs and the product of the inputs. The input parameters should not be modified. Use the principle of least privilege.

```
void computeSumProduct( ? x, ? y, ? sum, ? product ){
   sum = x + y;
   product = x * y;
}
int main(){
   int s, p, c = 10;
   double r, t = 3.2;
   computeSumProduct( ?, ?, ?, ? );









   return 0;
}
```

Exercise (returning more than one value):

Write a global function which takes two int parameters and computes and returns the sum of the inputs and the product of the inputs. The input parameters should not be modified. Use the principle of least privilege.

```
void computeSumProduct( const int x, const int y, int& sum, int& product ){
   sum = x + y;
   product = x * y;
}
int main(){
   int s, p, c = 10;
   double r, t = 3.2;
   computeSumProduct( c, t, s, p );

   // cannot pass a literal value, or an expression, or a variable of a different type to
   // a reference parameter (although it is ok to pass them to a pass-by-value parameter)
   // computeSumProduct( c + 4, 15, s + 5, r );   // COMPILE-TIME ERROR

   // order of evaluation of a function's arguments is not specified by the C++ standard
   // thus, different compilers can evaluate function arguments in different orders
   computeSumProduct( c++, c++, s, p );           // MAY YIELD A WARNING

   return 0;
}
```

Make objects printable with a global function

Overload the **<< operator** in order to print an object with **`cout`** easily

```
ostream& operator<<( ostream& out, const MyClass& myObject ) {
    // statements such as:
    // out << "[MyClass] property:" << myObject.getProperty();
    return out;
}
```

- The prototype of this function is included in the header file of the `MyClass` class but after (outside) the `MyClass` definition
- The implementation of this function is included in the source code file of the `MyClass` class but should be defined as a global function
- This function returns a reference to facilitate cascaded printing

Storage class, scope, and linkage

- Each identifier has several attributes
  - Name, type, size, and value
  - Also storage class, scope, and linkage
    - *Identifier's storage class* determines the period during which that identifier exists in memory (automatic and static variables)
    - *Identifier's scope* determines where the identifier can be referenced in a program
    - *Identifier's linkage* determines whether an identifier is known only in the source file where it is declared or across multiple files that are compiled, then linked together (use of the `extern` specifier)

Storage class (**automatic** variables)

- Only a local variable and a function parameter can be of the automatic storage class
  - Created when the program execution enters the block in which it is defined
  - Exists while the block is active
  - Destroyed when the execution exits the block
- Default declaration (can also be declared with keyword **`auto`**)

Automatic variables are stored in the call stack

What is the output?

```
void foo() {
   int k = 10;
   cout << k << endl;
   k++;
   if ( k > 5 ) {
      int a = 50;
      cout << a << endl;
      k++;
   }
   cout << k << endl;
}
int main() {
   int k = 20;
   cout << k << endl;
   k++;
   foo();
   cout << k << endl;
   return 0;
}
```

Storage class (**static** local variables)

- A local variable declared with **`static`**
  - Exists from the point at which the program begins execution
  - Initialized only once when its declaration is first encountered
  - Retains its value when the function (in which this local variable is declared) returns to its caller
    - Next time this function is called, it contains the value it had at the end of this function's last call
    - However, known only in this function (that is, its scope is only this function)

Static local variables are stored in the heap space

What is the output?

```
void g1() {
    int a = 4;
    cout << a << endl;
    a++;
}
void g2() {
    static int b = 4;
    cout << b << endl;
    b++;
}
int main() {
    g1();
    g1();
    g2();
    g2();
    // cout << a;  compile-time error
    // cout << b;  compile-time error
    return 0;
}
```

Storage class (**static** global variables)

- A global variable is static by its definition (keyword `static` is not used)
  - Defined by placing variable declaration outside any class or function definition
  - Created at the beginning of the program
  - Retains its value throughout the program execution
  - Can be referenced by any function that follows its declaration

  Do not use them as an alternative to parameter passing (do not use them if it is not that necessary!!!)

Global variables are also stored in the heap space

What is the output?

```
int k = 4;
void h1() {
   cout << k << endl;
   k = 7;
}
void h2() {
   cout << k << endl;
   k++;
}
void h3() {
   int k = 1;
   cout << k << endl;
   k = 20;
}
int main() {
   cout << k << endl;
   h1();          cout << k << endl;
   h2();          cout << k << endl;
   h3();          cout << k << endl;
   return 0;
}
```

Scope (**file** scope)

- Scope of an identifier: a portion of the program where it can be referenced
- Identifiers (global variables, global function definitions/prototypes) declared outside a function or a class all have the **file scope**
  - These identifiers can be referenced in all functions from the point at which they are declared until the end of the file

prog.cpp

```
double a1 = 3.14;
void f1(){ // can reference a1 (but not a2)
           // can call f1 (but not f2 or f3)
}
void f2(){ // can reference a1 (but not a2)
           // can call f1 and f2 (but not f3)
}
int a2 = 4;
void f3(){ // can reference a1 and a2
           // can call f1, f2 and f3
}
```

prog.cpp

```
// now every function
// can reference every global variable
// can call every global function
double a1 = 3.14;
int a2 = 4;
void f1();
void f2();
void f3();

void f1(){ ... }
void f2(){ ... }
void f3(){ ... }
```

Scope (**block** scope)

- Scope of an identifier: a portion of the program where it can be referenced
- Local variables and function parameters have the **block scope**
  - Block scope begins at the identifier's declaration
  - Block scope ends at the terminating } of the block in which the identifier is declared
  - Storage duration of the identifier does not affect its scope

```
void foo( int a, double b ) {
   int c;
   static double d = 3.14;
   Student S1;

   if (d > 1) {
      int e;
      Student S2;
   }
}
```

e and S2 have this block scope

a, b, c, d, and S1 have this block scope (their scopes begin at their declaration

An identifier in an outer block is "hidden" when a nested block has an identifier with the same name

What is the output?

```
int main() {
   int a = 5, b = 4;
   if (b > 0){
      int a = 2;
      cout << a << endl;
   }
   cout << a << endl;
   return 0;
}
```

Linkage

- Linkage of an identifier determines whether an identifier is known only in the source file where it is declared or across multiple files that are compiled and then linked together

- Identifier declared with the keyword `extern` will be linked with an identifier declared in another file with the same name

What is the output when a.cpp and b.cpp compiled and linked together?

a.cpp

```
#include <iostream>
using namespace std;
void bar() {
   extern int c;
   extern void foo();

   cout << c << endl;
   foo();
   cout << c << endl;
}
int main(){
   // cout << c;     compile time error
   // foo();         compile time error
   bar();
}
```

b.cpp

```
int c = 3;

void foo() {
   c = 5;
}
```

Functions with default arguments

- A default value may be defined for a function parameter
  - This default value will be used when the function call does not specify an argument for that parameter
  - Default values should be given to the rightmost argument(s) in a function's parameter list
  - Should be specified with the first occurrence of the function name (typically the function prototype)

What is the output?

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

int boxVolume( int , int = 1, int = 1 );

int main(){
   cout << boxVolume( 10, 5, 2 ) << endl;
   cout << boxVolume( 10, 5 ) << endl;
   cout << boxVolume( 10 ) << endl;

   // compile-time error since there is
   // no matching function for this call
   // cout << boxVolume() << endl;
    return 0;
}
int boxVolume( int W, int L, int H ){
    return W * L * H;
}
```

Overloaded functions

- These are the functions with the same name but a different signature (different parameter lists)
- The compiler selects a proper function to execute based on the number, types, and order of the arguments in a function call
- Commonly used to create several functions of the same name that perform similar tasks, but on different data types

Function templates

- More compact and convenient form of overloading
  - Identical program logic and operations for each data type

- Function template definition
  - Written by the programmer only once
  - Essentially defines a whole family of overloaded functions
  - Defined by `template < typename` *tname* `>` or `template < class` *tname* `>`

- Function template specializations will automatically be generated by the compiler to handle each type of call to the function template

Write a generic maximum function using a function template

main.cpp

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

#include "max.h"

int main() {
   cout << "The maximum of 5 and 3: ";
   cout  << maximum( 5, 3 ) << endl;

   double a = 4.5;
   cout << "The maximum of a and 5.6: ";
   cout  << maximum( a, 5.6 ) << endl;

   string s1 = "Hello", s2 = "World";
   cout << "The maximum of s1 and s2: ";
   cout  << maximum( s1, s2 ) << endl;

   // compile time error
   // cout  << maximum( a, 5 ) << endl;
   return 0;
}
```

max.h

```
#ifndef __MAX_H
#define __MAX_H

template < typename T >
T maximum( T num1, T num2 ) {
   if ( num1 > num2 )
      return  num1;
   else
      return num2;
}
#endif
```

Output

```
The maximum of 5 and 3:5
The maximum of a and 5.6: 5.6
The maximum of s1 and s2: World
```

The compiler generates a separate function definition for each call with arguments of a different type

C++ Standard Library header files

- The C++ Standard Library is divided into many portions, each with its own header file
  - Each header “instructs” the compiler on how to interface with its corresponding portion of the library

- Each header file contains
  - Prototypes for the related functions belonging to the portion of the library that the header file corresponds to
  - Definitions of various class types and constants necessary for its functions and for the client code

Standard Library header	Explanation
`<iostream>`	Contains function prototypes for the C++ standard input and output functions, introduced in Chapter 2, and is covered in more detail in Chapter 13, Stream Input/Output: A Deeper Look.
`<iomanip>`	Contains function prototypes for stream manipulators that format streams of data. This header is first used in Section 4.9 and is discussed in more detail in Chapter 13, Stream Input/Output: A Deeper Look.
`<cmath>`	Contains function prototypes for math library functions (Section 6.3).
`<cstdlib>`	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. Portions of the header are covered in Section 6.7; Chapter 11, Operator Overloading; Class `string`; Chapter 17, Exception Handling: A Deeper Look; Chapter 22, Bits, Characters, C Strings and `structs`; and Appendix F, C Legacy Code Topics.
`<ctime>`	Contains function prototypes and types for manipulating the time and date. This header is used in Section 6.7.
`<cctype>`	Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. These topics are discussed in Chapter 22, Bits, Characters, C Strings and `structs`.
`<cstring>`	Contains function prototypes for C-style string-processing functions. This header is used in Chapter 10, Operator Overloading; Class `string`.

Standard Library header	Explanation
`<array>`, `<vector>`, `<list>`, `<forward_list>`, `<deque>`, `<queue>`, `<stack>`, `<map>`, `<unordered_map>`, `<unordered_set>`, `<set>`, `<bitset>`	These headers contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The `<vector>` header is first introduced in Chapter 7, Class Templates `array` and `vector`; Catching Exceptions. We discuss all these headers in Chapter 15, Standard Library Containers and Iterators.
`<typeinfo>`	Contains classes for runtime type identification (determining data types at execution time). This header is discussed in Section 12.8.
`<exception>`, `<stdexcept>`	These headers contain classes that are used for exception handling (discussed in Chapter 17, Exception Handling: A Deeper Look).
`<memory>`	Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. This header is used in Chapter 17, Exception Handling: A Deeper Look.
`<fstream>`	Contains function prototypes for functions that perform input from and output to files on disk (discussed in Chapter 14, File Processing).
`<string>`	Contains the definition of class `string` from the C++ Standard Library (discussed in Chapter 21, Class `string` and String Stream Processing).
`<sstream>`	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 21, Class `string` and String Stream Processing).

Standard Library header	Explanation
`<functional>`	Contains classes and functions used by C++ Standard Library algorithms. This header is used in Chapter 15.
`<iterator>`	Contains classes for accessing data in the C++ Standard Library containers. This header is used in Chapter 15.
`<algorithm>`	Contains functions for manipulating data in C++ Standard Library containers. This header is used in Chapter 15.
`<cassert>`	Contains macros for adding diagnostics that aid program debugging. This header is used in Appendix E, Preprocessor.
`<cfloat>`	Contains the floating-point size limits of the system.
`<climits>`	Contains the integral size limits of the system.
`<cstdio>`	Contains function prototypes for the C-style standard input/output library functions.
`<locale>`	Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.).
`<limits>`	Contains classes for defining the numerical data type limits on each computer platform.
`<utility>`	Contains classes and functions that are used by many C++ Standard Library headers.

Math functions

- The <cmath> header file provides a collection of functions that enable you to perform common mathematical calculations
- All functions in the <cmath> header file are global functions—therefore, each is called simply by specifying the name of the function followed by parentheses containing the function's arguments

Function	Description	Example
ceil(x)	rounds *x* to the smallest integer not less than *x*	ceil(9.2) is 10.0 ceil(-9.8) is -9.0
cos(x)	trigonometric cosine of *x* (*x* in radians)	cos(0.0) is 1.0
exp(x)	exponential function e^x	exp(1.0) is 2.718282 exp(2.0) is 7.389056
fabs(x)	absolute value of *x*	fabs(5.1) is 5.1 fabs(0.0) is 0.0 fabs(-8.76) is 8.76
floor(x)	rounds *x* to the largest integer not greater than *x*	floor(9.2) is 9.0 floor(-9.8) is -10.0
fmod(x, y)	remainder of *x*/*y* as a floating-point number	fmod(2.6, 1.2) is 0.2
log(x)	natural logarithm of *x* (base *e*)	log(2.718282) is 1.0 log(7.389056) is 2.0
log10(x)	logarithm of *x* (base 10)	log10(10.0) is 1.0 log10(100.0) is 2.0
pow(x, y)	*x* raised to power *y* (x^y)	pow(2, 7) is 128 pow(9, .5) is 3
sin(x)	trigonometric sine of *x* (*x* in radians)	sin(0.0) is 0
sqrt(x)	square root of *x* (where *x* is a nonnegative value)	sqrt(9.0) is 3.0
tan(x)	trigonometric tangent of *x* (*x* in radians)	tan(0.0) is 0