Object-Oriented Programming
(in C++)

Functions

Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

Outline

- Introduction to Functions
- Functions (C vs. C++)
  - Some minor differences
  - Inline functions
  - Passing function arguments using references
  - Returning a reference
  - Default arguments
  - Function overloading
  - Function templates
- Storage Classes
- Scope Rules

Functions

- A function is a collection of statements that performs a specific task---a single, well-defined task
- Divide and conquer technique
  - Allow programs to be broken down into smaller pieces
- This makes programs easier to read and maintain
  - Process abstraction
  - Statements in function bodies are written only once
    - Reused from perhaps several locations in a program
    - Hidden from other functions
    - Avoid repeating code
- Function call and usage is the same as in C (but ANSI-style only!)

Function Definition

```
return-value-type function-name(parameter-list)
{
  declaration and statements
}
```

- `return-value-type`
  - Data type of the result returned by the function
- `function-name`
  - A valid identifier
- `parameter-list`
  - A comma-separated list containing the declarations of the parameters passed to the function
- `Declaration and statements`
  - Function body

```
int foo(int) {
  ...
}
```

Function Prototype

- Also called a function declaration
  - Function prototype is provided before the function is invoked
  - E.g., `int foo( int );`
- Indicates to the compiler:
  - Function signature
    - Name of the function
    - Parameters the function expects to receive
      - Number of parameters
      - Types of those parameters
      - Order of those parameters
  - Type of data returned by the function
- If a function is defined before it's invoked, its definition also serves as the function's prototype

Function: An Example

```
// Fig. 5.3: fig05_03.cpp
// Creating and using a programmer-defined function.
#include <iostream>
using namespace std;

int square( int ); // function prototype

int main()
{
   // loop 10 times and calculate and output the
   // square of x each time
   for ( int x = 1; x <= 10; x++ )
      cout << square( x ) << "  "; // function call

   cout << endl;
} // end main

// square function definition returns square of an integer
int square( int y ) // y is a copy of argument to function
{
   return y * y;     // returns square of y as an int
} // end function square
```

```
1  4  9  16  25  36  49  64  81  100
```

Outline

- Introduction to Functions
- Functions (C vs. C++)
  - Some minor differences
  - Inline functions
  - Passing function arguments using references
  - Returning a reference
  - Default arguments
  - Function overloading
  - Function templates
- Storage Classes
- Scope Rules

Return Type: C vs C++

- C
  - If no return type is specified, the function is declared to return a value of type `int`
  - The void return type does not need to be specified if a return statement is missing

```
foo() {
  printf("Hello");
}
```

C

```
void foo() {
  cout << "Hello";
  return;
}
```

C++

- C++
  - If no value is returned in the function definition, C++ requires the void return type

Function Prototype: C vs C++

- If parameters are not specified in the function's prototype
  - C
    - Any number of values (or no value) can be passed to the function when it is called
  - C++
    - No value can be passed to the function when it is called

```
void foo(); // function prototype
...
foo(a, b);  // no problem in C
            // an error in C++
```

Type Casting and Coercion (C ≅ C++)

- Casting: explicit type conversion
  - By programmer
  - E.g., `bool SomeBool = (bool)SomeInt;`
- Coercion: implicit type promotion
  - By compiler
  - E.g., `double SomeDouble = SomeInt;`

Data types	
long double	
double	
float	
unsigned long int	(synonymous with unsigned long)
long int	(synonymous with long)
unsigned int	(synonymous with unsigned)
int	
unsigned short int	(synonymous with unsigned short)
short int	(synonymous with short)
unsigned char	
char	
bool	

Coercion goes in only this direction

Fig. 5.5 | Promotion hierarchy for fundamental data types.

Global vs Member Functions

- Global functions
  - Functions that are not members of a class
  - For procedural programming (as in C)
- Member functions (only in C++)
  - Functions that are members of a class
  - For objet-oriented programming
  - Often called “methods” in other OOP languages

```
Class foo {
  int bar() {  // member function
    ...
  }
}
int main() {   // global function
  ...
}
```

Functions: C vs C++

- Function call and usage is the same as in C (but ANSI-style only!)
- Extra features of function usage in C++ include :
  - Inline functions
  - Passing function arguments using references
  - Returning a reference
  - Default arguments
  - Function overloading
  - Function templates

Inline Functions

- C++ provides inline functions to help reduce function call overhead—especially for small functions
- To “advise” the compiler to generate a copy of the function’s code in place (when appropriate) to avoid a function call
  - The compiler can ignore the inline qualifier and typically does so for all but the smallest functions

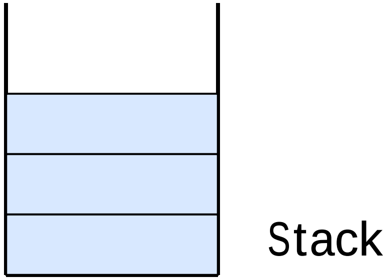
```
inline int cube(int s) {
  return s * s * s;
}
int main() {
   ...
   i = cube(x);
   ...
   j = cube(y);
}
```

```
int main() {
   ...
   i = x * x * x;
   ...
   j = y * y * y;
}
```

- The trade-off
  - Performance vs code size expansion

Function Call Stack and Activation Records

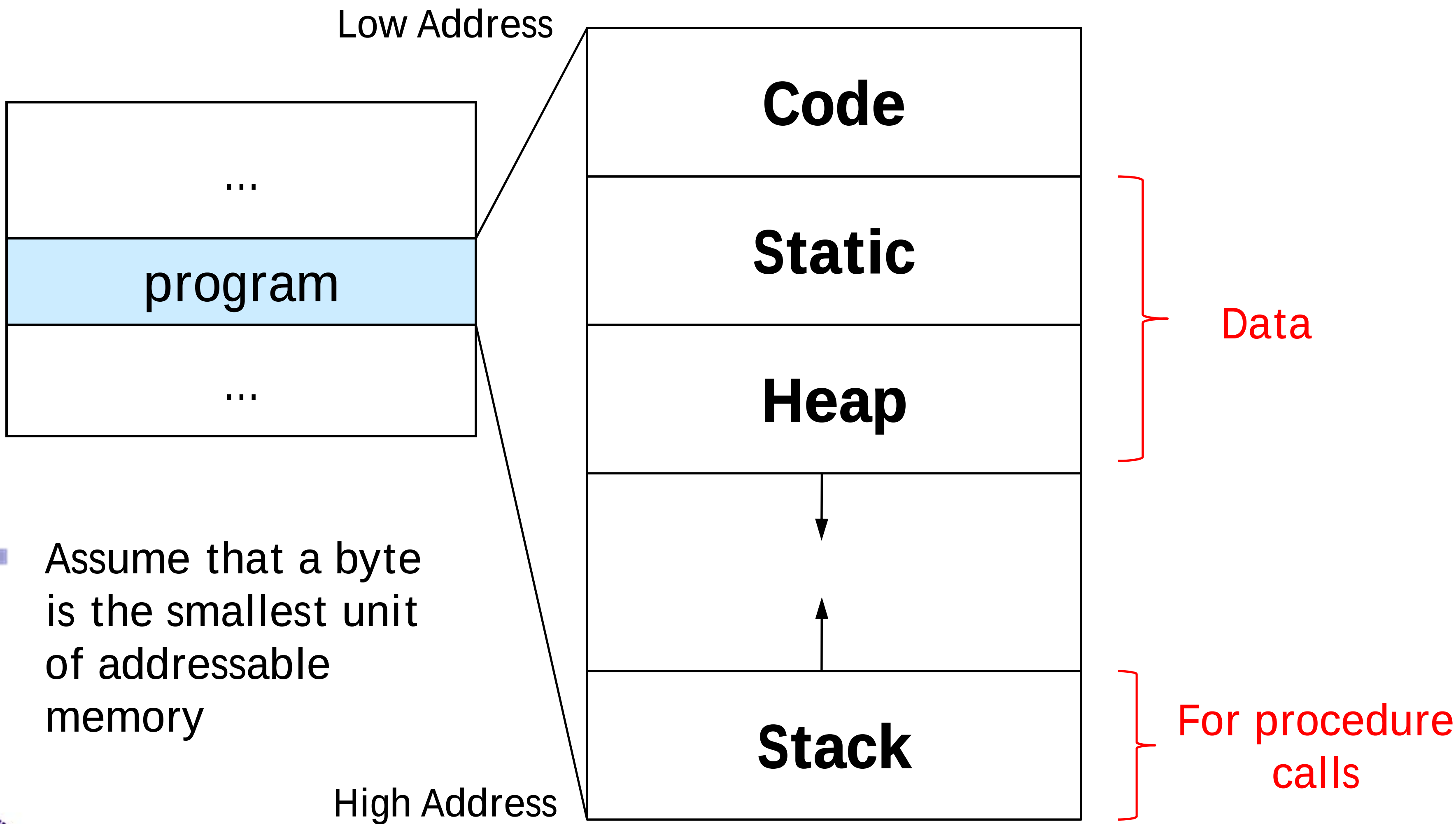
- To understand how C++ performs function calls, we first need to consider a data structure known as a stack
  - Think of a stack as analogous to a pile of dishes
  - When a dish is placed on the pile, it's normally placed at the top (referred to as pushing the dish onto the stack)
  - Similarly, when a dish is removed from the pile, it's normally removed from the top (referred to as popping the dish off the stack)



- Stacks are known as last-in, first-out (LIFO) data structures—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack

Supplement: Storage Layout

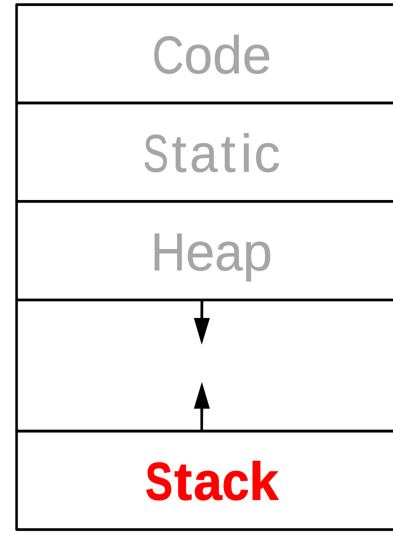
- Typical memory layout of a program



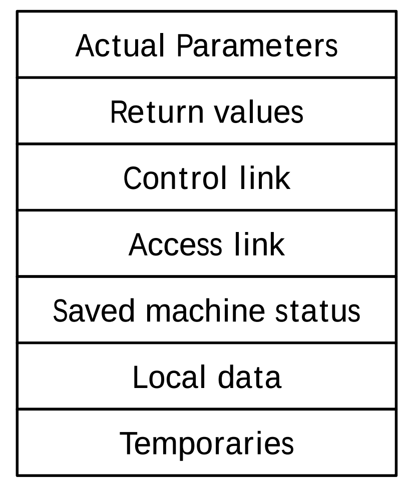
- Assume that a byte is the smallest unit of addressable memory

Supplement: Activation Records

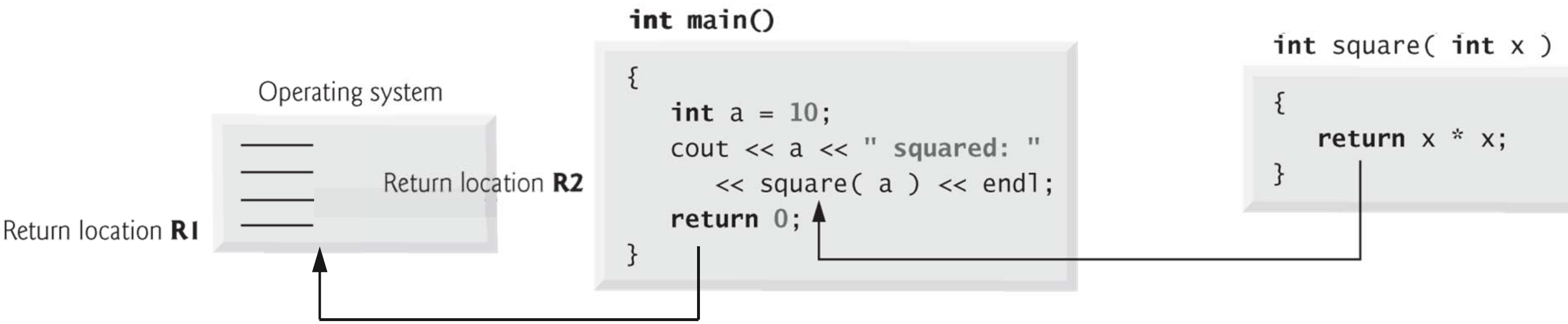
- Stack is used to store activation records that get generated during procedure calls



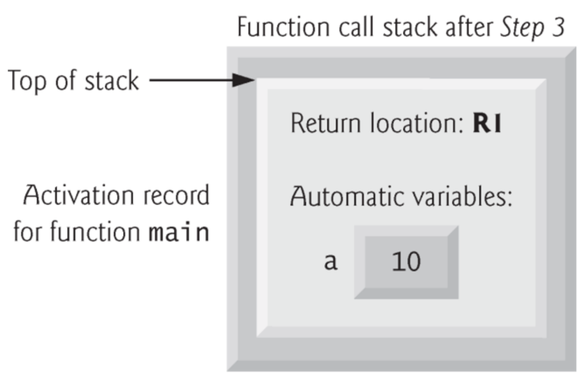
- Activation record (frame)
  - Used to store information about
    - The status of the machine, such as the value of the program counter and machine registers, when a procedure call occurs
    - Data objects whose lifetime are contained in that of an activation
    - ...

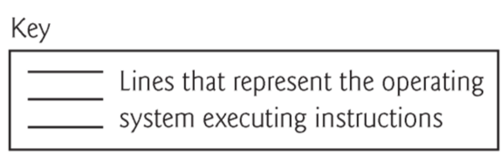


How Does A Function Call Work?



1. Push `main`'s AR, save return location R1, and jump to `main`
2. Push `square`'s AR, save return location R2, and jump to `square`
3. Return to R2 and pop
4. Return to R1 and pop





Inline vs Macro

```
inline int cube(int s) {
  return s * s * s;
}
int main() {
  ...
  i = cube(x);
  ...
  j = cube(y);
}
```

```
#define cube(s) s * s * s


int main() {
  ...
  i = cube(x);
  ...
  j = cube(y);
}
```

- This process could be defined as macro, but it has side effect
  - E.g., `cube(x++)` produces `x++ * x++ * x++`

Passing Function Arguments by Reference

- In C++ (and C) function arguments are passed by value or by reference
- Pass by reference in C is implemented through the use of pointers
- A side effect is that the function may change the value of its actual argument on exit

Pass-by-Reference: Pointer vs Reference

```
void func(int arg1, int* arg2) {
  // arg1 is passed by value
  // arg2 is passed by reference
  arg1++;
  (*arg2)++;
}

void main() {
  int i,j;
  i = j = 0;
  func(i, &j);

  // i=0, j=1
  return 0;
}
```

```
void func(int arg1, int& arg2) {
  // arg1 is passed by value
  // arg2 is passed by reference
  arg1++;
  arg2++;
}

int main() {
  int i,j;
  i = j = 0;
  func(i, j);

  // i=0, j=1
  return 0;
}
```

Simpler syntax (don't need to remember to de-reference formal arguments inside the function)

Returning A Reference

- A function can return a reference to an object
- Allows function calls to be used as *lvalues*
  - They can appear on the left hand side of assignments
  - This is a nice programming trick as long as we know what we are doing

Returning A Reference: An Example

```
int a[20];

int& access(int index) {
  static int tmp = 0;

  if ((index>=0) && (index<20))
    return a[index];
  else
    return tmp;
}

int main() {
  int val=access(7); // val=a[7]
  access(7)=20;      // a[7]=20
  access(20)=20;     // tmp=20

  return 0;
}
```

Default Arguments

- A default value to be passed to a parameter
  - Used when the function call does not specify an argument for that parameter
- Must be 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
  - It is a compilation error to specify default arguments in both a function's prototype and header

Default Arguments: An Example

```cpp
// Fig. 5.21: fig05_21.cpp
// Using default arguments.
#include <iostream>
using namespace std;

// function prototype that specifies default arguments
int boxVolume( int length = 1, int width = 1, int height = 1 );

int main()
{
   // no arguments--use default values for all dimensions
   cout << "The default box volume is: " << boxVolume();

   // specify length; default width and height
   cout << "\n\nThe volume of a box with length 10,\n"
      << "width 1 and height 1 is: " << boxVolume( 10 );

   // specify length and width; default height
   cout << "\n\nThe volume of a box with length 10,\n"
      << "width 5 and height 1 is: " << boxVolume( 10, 5 );

   // specify all arguments
   cout << "\n\nThe volume of a box with length 10,\n"
      << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
      << endl;
} // end main

// function boxVolume calculates the volume of a box
int boxVolume( int length, int width, int height )
{
   return length * width * height;
} // end function boxVolume
```

boxVolume(1, 1, 1)

boxVolume(10, 1, 1)

boxVolume(10, 5, 1)

Function Overloading

- C++ allows the same name to be used for two or more different functions
- Requirements for overloaded functions
  - They have different signatures
    - i.e., different ordered parameter types

```
int foo(int) {...}
int foo(double) {...}
int foo(int, double) {...}
```

```
int foo(int, char) {...}
int foo(char, int) {...}
```

  - It is not possible to overload functions by changing their return types

```
int foo(int) {...}
char foo(int) {...}
```

  - Why?

Ambiguity When Using Function Overloading

```
void fun1(float) {...}
void fun1(double) {...}
void fun2(int=1) {...}
void fun2() {...}

int main() {
  int x = 3;
  float y = 4.4;
  double z = 5.5555;

  fun1(y);  // valid: fun1(float) is called
  fun1(z);  // valid: fun1(double) is called
  fun1(x);  // ambiguity: x can be coerced to float or double
  fun2(x);  // valid: fun2(int) is called
  fun2();   // ambiguity: default argument can be used

  return 0;
}
```

Function Templates

- Overloaded functions are normally used to perform similar operations that involve different program logic on different data types
- If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently by using function templates

```
int square(int x) {
  return x * x;
}
double square(double x) {
  return x * x;
}
```

```
template <class T>
//template <typename T>
T square(T x) {
  return x * x;
}
```

  - Defining a single function template essentially defines a whole family of overloaded functions

Outline

- Introduction to Functions
- Functions (C vs. C++)
  - Some minor differences
  - Inline functions
  - Passing function arguments using references
  - Returning a reference
  - Default arguments
  - Function overloading
  - Function templates
- Storage Classes
- Scope Rules

Storage Classes

- Storage classes are modifiers that change how the memory associated with the variable behaves within a function
  - There are five storage classes (`auto`, `register`, `static`, `extern`, `mutable`)

    C++ only

    - `auto`
      - Allocate the variable when it is declared
      - Deallocate it when it is no longer in scope
      - This applies to local variables
    - `register`
      - Used with an automatic variable to suggest to the compiler that this variable be kept in a hardware register
      - Automatic and register variables that are not initialized will have undefined values
    - `static`
      - This causes a variable to retain its value throughout the execution of the program
      - The value is zero if not explicitly initialized, unless it is a pointer

```
auto int flag;
register int flag;
static int flag
```

Static vs Automatic Local Variables

This example is modified from Fig. 6.12

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

void staticVarInit() {
  static int var1; // initialized to 0 first time it is called
  cout << "value on entering staticVarInit:" << var1 << endl;
  var1++; // increase the static variable by one
  cout << "value on exiting staticVarInit:" << var1 << endl;
}

void autoVarInit() {
  int var2 = 0;
  cout << "value on entering autoVarInit:" << var2 << endl;
  var2++; // increase the automatic variable by one
  cout << "value on exiting autoVarInit:" << var2 << endl;
}

int main() {
  cout << "First call to each function" << endl;
  staticVarInit();
  autoVarInit();

  cout << "\nSecond call to each function" << endl;
  staticVarInit();
  autoVarInit();
}
```

```
First call to each function
value on entering staticVarInit:0
value on exiting staticVarInit:1
value on entering autoVarInit:0
value on exiting autoVarInit:1

Second call to each function
value on entering staticVarInit:1
value on exiting staticVarInit:2
value on entering autoVarInit:0
value on exiting autoVarInit:1
```

Storage Classes (Cont'd)

- `extern` and `static` have special meaning when they are applied explicitly to external identifiers (global variables or global function names)
  - ```
    extern int flag;
    ```
    - Indicates that `flag` is defined either later in the same file or in a different file
  - ```
    static double pi = 3.14;
    ```
    - Indicates that `pi` is know only to functions in the file in which it is defined

Outline

- Introduction to Functions
- Functions (C vs. C++)
  - Some minor differences
  - Inline functions
  - Passing function arguments using references
  - Returning a reference
  - Default arguments
  - Function overloading
  - Function templates
- Storage Classes
- Scope Rules

Scope Rules

- Scope: portion of the program where an identifier can be used
  - i.e., the scope of a variable is the range of statements over which it is visible
- We discuss four scopes for an identifier
  - Global namespace scope (global scope)
  - Local scope (block scope)
  - Function-prototype scope
  - Function scope
    - Labels are the only identifiers
- Two other scopes—class scope (Chapter 9) and namespace scope (Chapter 24)

Scope Rules: Global Scope

- A name has global namespace scope if the identifier's declaration appears outside of all blocks, namespaces, and classes
- Global variables, function definitions and function prototypes placed outside a function all have global namespace scope

Scope Rules: Local Scope

- Identifiers declared inside a block have block scope
  - Block scope begins at the identifier's declaration
  - Block scope ends at the terminating right brace (}) of the block in which the identifier is declared
- Local variables and function parameters have block scope
  - The function body is their block

Scope Rules: Function-Prototype Scope

- The only identifiers with function-prototype scope are those used in the parameter list of a function prototype
- Function prototypes do not require names in the parameter list—only types are required
  - Names appearing in the parameter list of a function prototype are ignored by the compiler
  - Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity

Unary Scope Resolution Operator

- C++ provides the unary scope resolution operator (::) to access a global variable when a local variable of the same name is in scope

```
// Fig. 5.22: fig05_22.cpp
// Using the unary scope resolution operator.
#include <iostream>
using namespace std;

int number = 7; // global variable named number

int main()
{
   double number = 10.5; // local variable named number

   // display values of local and global variables
   cout << "Local double value of number = " << number
      << "\nGlobal int value of number = " << ::number << endl;
} // end main
```

```
Local double value of number = 10.5
Global int value of number = 7
```