CS 2351 Data Structures

Basics of C++

Prof. Chung-Ta King
Department of Computer Science
National Tsing Hua University

C and C++

- C and C++ are closely related
  - C++ grew out of C and was designed to be source-and-link compatible with C
- C++ is evolving
- C++ is often considered to be a superset of C
  - Most C code can be made to compile correctly in C++, but some valid C code are invalid or behave differently in C++
- C++ is described as "a better C"
  - C++ supports OOP and more; but C++ is not pure OOP, i.e. you can write non-OO programs (C-like) using C++
  - Study basics here and leave OOP and other features later

Outlines

- Program organization
- Scope and namespace
- Declaration of variables
- Functions
  - Parameter passing, function overloading, inlining
- Dynamic memory allocation
- Exceptions

Basic Program Structure: “Hello, World!”

C:

```
#include <stdio.h>
int main(void)
{
    printf("Hello, world!\n");
    return 0;
}
```

C++:

```
// Hello, World! in C++
#include <iostream>
int main(void)
{
   std::cout<<"Hello, world!”<<std::endl;
   return 0;
}
```

Some Notes from C++ "Hello, World!"

- Comments:
  - One line comment:

```
// Hello, World! in C++
```

  - Multiple line comment:

```
/* Hello, World!
     in C++ */
```

- `#include <iostream>`
  - Instruct the preprocessor to include C++ *header iostream*, that performs standard input and output operations

Some Notes from C++ “Hello, World!”

- **`std::cout`**
  - Identifies the *standard character output device* (usually, computer screen)
  - For input, use **`std::cin`**
    `std::cin >> a >> b;`
- **<<**
  - *Insertion operator*: indicate that what follows is inserted into **`std::cout`**
- **`std::endl`**
- File I/O by including head file *fstream* and defining a filestream variable: ofstream *outFile*(“abc”,ios::out);

Some Notes from C++ "Hello, World!"

- Use **`cout`** instead of **`std::cout`**
  - **`cout`** is part of the standard C++ library, where all the elements are declared within the *namespace* **`std`**
  - These elements may be referred to either *qualified* (e.g. **`std::cout`**) or made visible by the *using* declaration:

```
#include <iostream>
using namespace std;
int main ()
{
  cout << "Hello World! " << endl;
}
```

This allows all elements in **`std`** namespace to be accessed in an *unqualified* manner (without the **`std::`** prefix)

Namespace Scope

- Group related variables and functions together into narrower logical scopes → avoid name collision

```
namespace foo {
   int value() { return 5; }
}
namespace bar {
   const double pi = 3.1416;
   double value() { return 2*pi; }
}
   ...
   cout << foo::value() << '\n';
   cout << bar::value() << '\n';
   cout << bar::pi << '\n';
```

Keyword “using”

- The “using” keyword can directly expand namespace

```
namespace first {
   int x = 5;          int y = 10;
}
namespace second {
   double x = 3.1416; double y = 2.7183;
}
   ...
   using first::x; using second::y;
   cout << x << '\n';
   cout << y << '\n';
   cout << first::y << '\n';
   cout << second::x << '\n';
```

4 Types of Scopes in C++

Each variable has a scope and is uniquely identified by its scope and its name. A variable is visible to a program only from within its scope.

- Local scope:
  - A name declared in a block is in local scope of that block

```
void func1() {
    int i;
    for (i=0; i<10; i++) {
        int j = 42;
        printf("%d %d\n", i, j);
    }
}
```

4 Types of Scopes in C++

- Namespace scope:
  - Discussed above
- Class scope:
  - Declarations associated with a class definition; each class represents a distinct class scope; discussed next chapter
- File scope:
  - Declarations not contained in a function definition, class definition, or a namespace

Outlines

- Program organization
- Scope and namespace
- Declaration of variables
- Functions
  - Parameter passing, function overloading, inlining
- Dynamic memory allocation
- Exceptions

Data Declaration

- Data declaration associates a data type with a name
  - Constant values: 5, ‘a’, 4.3
  - Variables
  - Constant variables: variables cannot be assigned a value
  - Enumeration types:
    enum *semester* {SUMMER, FALL, SRPING};
  - Pointers
  - Reference types: an alternative name for an object
    `int i = 5;`   `int& j = i;`
    when i’s value is changed, j’s value changes correspondingly

Reference Variables

- Reference = alias
  - The operator "&" has been extended in C++

```
int          id = 100;
int         *id_ptr = &id;
const int   *cid_ptr = &id;
int         &id_alias = id;
const int   &cid_alias = id;
```

  - Now **id** and **id_alias** are bound to the "same" variable

Reference and Pointer

- Pointer can be NULL, but reference CANNOT be NULL (reference must be bound to a variable)

```
int *ptr = NULL;    // address = 0
int &ptr = NULL;    // syntax error
```

- Binding target of reference CANNOT be changed

```
int  y = 20;
ptr = &y; // pointer can change target
```

- Pointers can be initialized at any time, but a reference must be initialized when it is created

Outlines

- Program organization
- Scope and namespace
- Declaration of variables
- Functions
  - Parameter passing, function overloading, inlining
- Dynamic memory allocation
- Exceptions

Functions in C++

- Two types of functions:
  - Regular functions
  - Member functions: associated with C++ classes
- Function components:
  - Name, arguments (*signature*), return type, body
- Function declaration (function prototype):

```
int add(int, int);
```

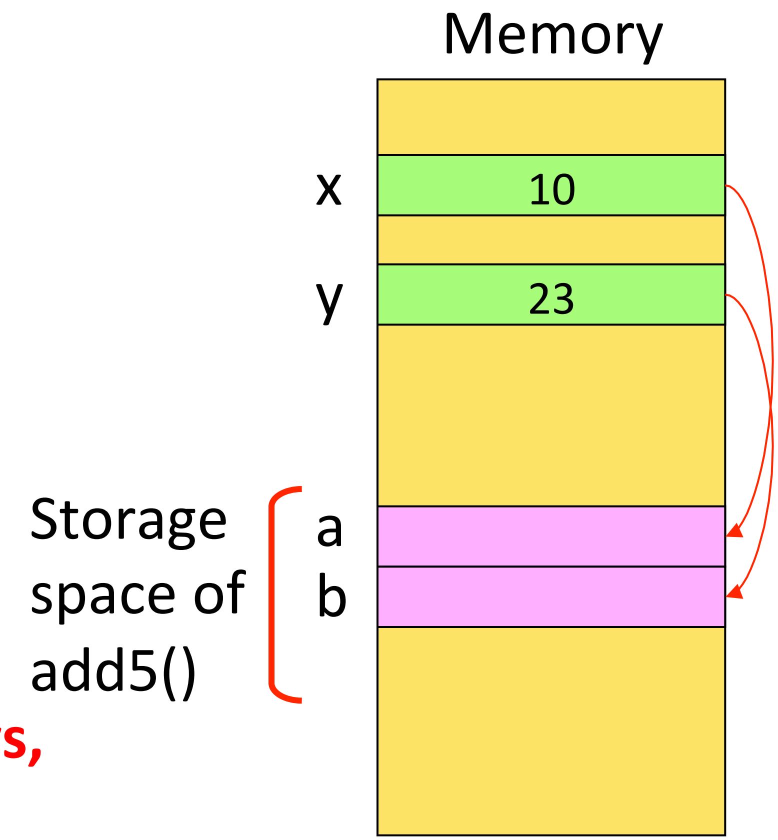
- Function definition:

```
int add(int a, int b){
        return a+b;
}
```

http://www.cplusplus.com/doc/tutorial/functions/

Parameter Passing: Call-by-Value

- When an object is passed by *value*, it is copied into the function's local storage and the function accesses its local copy.



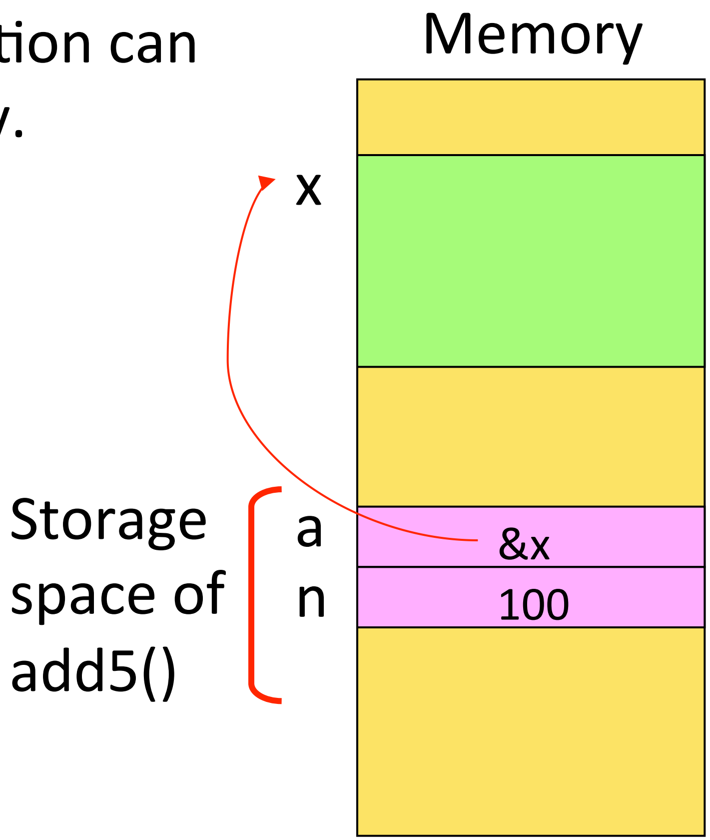

```
int add5(int a, int b)
{
      a = a + 5;
      return a + b;
}
        ...
      add5(x, y);
```

What happen if arguments are arrays, e.g., a[100] and b[100]?

What If "Value" Is "Address"?

- Recall in C, we can use pointers as arguments
  - When the "value" of a pointer variable is passed, an address is passed and the function can access the actual object directly.

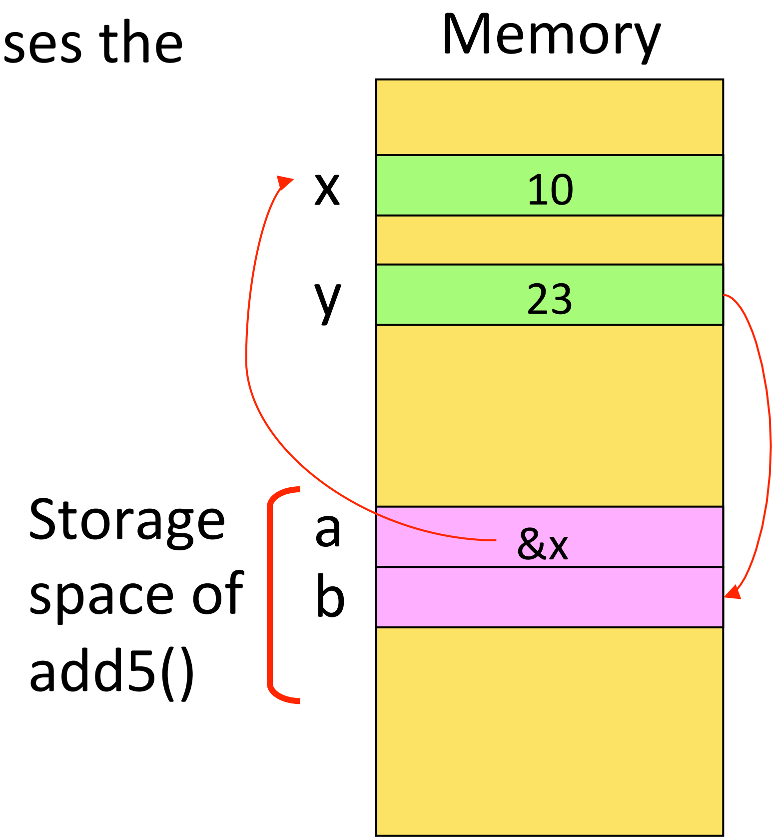
```
void addv(int *a, int n)
{
      a[n-1] = a[0] + 5;
      return 0;
}
      int x[100];
      ...
      addv(x, 100);
```



Reference Variables for Call-by-Reference

- In C++, an argument may be passed by *reference*
  - Only the address of the object is copied to the function's local store, and function accesses the actual arguments
  - Default for array types

```
int add5(int& a, int b)
{
      a = a + 5;
      return a + b;
}
       ...
      add5(x, y);
```



Function Overloading

- In C++, we can define functions with same name but different *signatures* in the same program, e.g.

```
int    Max(int, int);
int    Max(int, int, int);
int    Max(int*, int);
int    Max(float, int);
```

- In C, it is impossible to define two functions with same function name

How Function Overloading Work in C++ ?

- Function signature is defined in C using
  - Function name
- Function signature is defined in C++ using
  - Function name
  - Type of parameters
  - Order of parameters

Inline Functions

- An inline function is declared by keyword **inline**
  - Compiler will replace all calls to the function by its body → eliminate overhead of function calls/returns

```
inline int Sum(int a, int b)
{
     return a + b;
}
```

Outlines

- Program organization
- Scope and namespace
- Declaration of variables
- Functions
  - Parameter passing, function overloading, inlining
- Dynamic memory allocation
- Exceptions

Dynamic Memory Allocation

- new and delete operators
  - An object created by **new** exists for the duration of the program unless it is explicitly deleted by **delete**
  - In C, dynamic memory allocation is done through library functions **malloc()** and **free()**

```
#include <iostream>
#include <cstdio>
   int *x  = (int*) malloc(sizeof(int));
   free(x);
   int * y =  new int ;
   delete y ;
   int * data  = new int [10];
   delete  [] data ;
```

const vs. #define

- “const”: new keyword to declare constant variables

```
int main() {
    const int SIZE = 5;

    SIZE = 10; //  compiler  ERROR
}
```

- Compiler will do type-check for you. The #define macro cannot achieve this.

Outlines

- Program organization
- Scope and namespace
- Declaration of variables
- Functions
  - Parameter passing, function overloading, inlining
- Dynamic memory allocation
- Exceptions

Exception Handling

- Exceptions are used to signal occurrences of run -time errors and other special conditions
  - Hardware may signals exceptions
  - C++ programs can check for exceptional conditions and throw an exception

```
int DivZero(int a, int b, int c)
{
   if (a <= 0 || b <= 0 || c <= 0)
      throw "All parameters should be > 0";
   return a + b / c;
}
```

Exception Handling

- Exceptions that might be thrown by a piece of code can be handled by enclosing this code within a **try** block, followed by zero or more **catch** blocks
  - The **catch** block has an argument whose type determine the type of exception caught by that **catch** block
  - A **catch** block typically contains code to recover from the exception that has occurred
  - When an exception is thrown, normal execution of the **try** block terminates and the first **catch** block that matches the type of the thrown exception is executed, with the remaining **catch** blocks bypassed

Exception Handling

```
#include <iostream>         // std::cerr
#include <exception>        // std::exception
int main () {
  try {
     if( hasError() ){
        throw  20;
     }
  } catch ( int ERRNO ){  // catch exception int
    std::cerr << "ERRORNO=" << ERRNO << '\n';
  } catch ( ... )  {  //catch all types of exceptions
       std::cerr << "exception caught: " << '\n';
  }
  return 0;
}
```

Summary

- C++ is a better C
  - In addition to OOP, C++ provides many new features to facilitate programming: reference variables, cout/cint, namespace, call-by-reference, function/operator overloading, inline function, exception handling, …
- Further readings:
  - http://www.cplusplus.com/doc/tutorial/
  - Any textbook on C++
  - MIT's Introduction to C++
    http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-096-introduction-to-c-january-iap-2011/