CS 520 Advanced Programming Languages
Fall Semester, 2009
Doc 7 C++ Functions
Sept 23, 2009

References

C++ Primer, Stanley Lippman

1994 CS 535 Lecture notes, private document

Forward Declaration of Function

```
#include <iostream>

int min(int a, int b);
int max(int, int);                  // no parameter names needed
int goofy(int why, int notMe);      // trouble ahead

int main() {
        std::cout << min(2, 5) << "\n";
        goofy(10,5);
}
```

```
int min(int a, int b) {
        return  a < b ? a : b;
}

int max(int a, int b) {
        return  a > b ? a : b;
}

int goofy(int notMe, int why) {
        std::cout << "This is legal, not = " << notMe << std::endl;
        return why;
}
```

Parameter Passing

```
#include <iostream>
// pass-by-value
void swap (int a, int b) {
        int temp = b;
        b = a;
        a = temp;
}

void pointerSwap (int *a, int *b) {
        int temp = *b;
        *b = *a;
        *a = temp;
}

// pass-by-reference
void referenceSwap( int &a, int &b) {
        int temp = b;
        b = a;
        a = temp;
}
```

```
int main()
{
        int x = 10, y = 20;
        swap(x, y);                     // no effect
        pointerSwap(&x, &y);            // this works
        referenceSwap(x, y);            // this also works
}
```


pointer pass by value = pass by reference

Return-by-value

```
int isThisClear()  {
        int local = 1;
        return local;
}

int main() {
      int why;

      why = isThisClear() + 2;
      std::cout << why;
}
```

Return-by-reference

```
int& thisIsDangerous()  {
        int local = 1;
        return local;
}

int main() {
        int why;
        why = thisIsDangerous() + 2;
        std::cout << why;
}
```


Generates compiler warning

Pointers & Return-by-reference

```
int& pointersWork(int& in )  {
        int local = 5;
        int* localPointer = new int(8);

        if (in < 3)
                return local;                   // bad news
        else
                return *localPointer;          // ok
}
```

Reference Fun

```
#include <iostream>

int& isThisConfusing(int& in )  {
        in = in + 1;
        return in;
}
```

What is the output?

```
int main() {
        int why;
        int notThis = 3;

        why = isThisConfusing(notThis) + notThis;
        std::cout << why;
}
```

Variable number of Arguments

```
#include <iostream>
#include <stdarg.h>

// This sums up a list of integers. The last integer must be zero.

long int sum( int MyArg,...)
{
        va_list argumentList;
        int next;
        long int total = MyArg;

        va_start( argumentList, MyArg);     // Must call va_start,
        while (next=va_arg(argumentList,int))
                total += next;
        va_end( argumentList );             // Needed for clean up
        return( total );
}

int main(){
        long int result;
        result = sum( 2, 4, 6, 8, 0 );
        std::cout << "The sum of 2, 4, 6, and 8 is " << result;
}
```

Default Parameters

```
#include <iostream>

int myTest( int a, int b = 1, char c = 'a') {
        return ( a + b + c);
}

int main()
{
        int x;

        x = myTest(4);                  //102
        x = myTest(3, 5);               //105
        x = myTest(2 ,9 , 'x');         //131
        x = myTest(2, 'x');                     //What do you think?
}
```

With Forward declaration

```
int min(int a, int b = 10);


int min(int a, int b) {                 // no default needed here
      return  a < b ? a : b;
}
```

Overloading functions

```
void overHere(int a) {
        std::cout << "Who's on first?\n";
}

void overHere(float a) {
        std::cout << "Where is second?\n";
}
```

```
int main(){
        int a;
        float b;
        char c;
        overHere(a);        // prints  Who's on first?
        overHere(b);        // prints  Where is second?
        overHere(c);        // prints  Where is second?
}
```

Rules for Finding Correct Overloaded Function

One Argument

Look for exact match between formal and actual arguments

Promote actual argument and look for match

Actual	Formal
char, short, unsigned char	int
unsigned short	int (if int > short) unsigned int (otherwise)
float	double
enumerated	int

Match by Standard Conversion

numeric type	any numeric type
enumerated	any numeric type
zero	pointer or numeric type
pointer	void*

Rules for Finding Correct Overloaded Function

Multiple Arguments

The function that

Match of each argument is the same or better than for all other functions of the same name

Match must be better than all other functions for of one argument

A function is ambiguous if

No one function instance contains a better match or

More than one function instance contains a better match

Examples

```
void itsMe( char*, int );
void itsMe( int, int );

itsMe( 0, 'a' );
```

```
void dont( long,  long );
void dont( double, double );

int give, up;
dont( give, up );
```

```
void you( int, int );
void you( float, float );

int fooled;
float me;

you( fooled, me );
```

Return value does not count

```
int trustMe(int dont) {
        cout << "Big" << endl;
        return 5;
}

float trustMe(int doIt) {                    // compile error
        cout << "Mistake" << endl;
        return 5.5;
}


int main() {

        float whichOne;
        whichOne = trustMe(2);

}
```

Default Values and Matching

```
int trustMe(int dont, float doIt = 5.5) {
        return 5;
}

int trustMe(int dont, int doIt = 5) {
        return 10;
}

int main() {
        int whichOne;

        whichOne = trustMe(2, 3);              // OK

        whichOne = trustMe(2);          // compile error
}
```

Inline Functions

```
inline int BadMax(int x, int y) {
        return (x > y ? y : x);
}

inline void printMe(int x) {
        cout << x << "\n";
}
```

```
int main() {
        int a = 5 , b = 3;

        a = BadMax(a,b);
        printMe(a);
}
```

```
int main(){
        int a = 5 , b = 3;

        a =  a > b ? b : a ;
        cout << a << "\n";
}
```


inline is only a hint to the compiler

Template (Generic) Functions

```
template <class MyType>
MyType minTest( MyType a, MyType b) {
        return a < b ? a : b;
}

int main() {
        int a = 1, b = 2;
        float c = 1.1, d = 2.2;
        int e[10], f[10];
        cout    << minTest(a,b)         // prints 1
                << "\n"
                << minTest(c,d)         // prints 1.1
                << "\n"
                << minTest(e, f)  // prints 0xbfffe790
                << '\n'
                << minTest(a,c);  // compile error
}
```

Scope and ::

```
int where = 10;

int main() {

        int where = 5;
        cout
                << ::where    // prints 10
                << "\n"
                << where                // prints 5
                << "\n";
}
```

Function Pointers

```
void quickSort( int* array, int LowBound, int HighBound) {// source code removed  }

void mergeSort(int* array, int LowBound, int HighBound) {// same here }

void insertionSort(int* array, int LowBound, int HighBound){// ditto  }

int main() {
      void (*sort) (int*, int, int);
      int Size;
      int Data[100];                  // pretend data and Size are initialized

      if (Size < 25)
            sort = insertionSort;
      else if (Size > 100)
            sort = quickSort;
      else
            sort = mergeSort;
      sort(Data, 0, 99);
}
```

Arrays of Function Pointers

```
void doNothing()        { blah }
void haveFun()          { blah }
void trouble()          { blah }
void check()        { blah }

int getMenuItemSelected() {
    // blah
    return 0;
}
```

```
int main() {
    void (*menuActions[]) () = {
        doNothing,
        haveFun,
        check,
        trouble
    };
    menuActions[getMenuItemSelected()];
}
```

Memory Issues

```
int main() {
        int *noWhere;
        *noWhere = 5;    //bus error
}
```

New & Delete

```
int main() {
        int *noWhere;

        noWhere = new int;          // allocate space

        *noWhere = 5;

        cout << *noWhere << "\n";

        delete noWhere;             // deallocate space
}
```

Function Pointers & New

```
#include <iostream>
#include <new>                  // For set_new_handler

void outOfMemory() {
      std::cerr << "Ran out of memory. Good Bye!"  << std::endl;
      exit(1);
};

int main() {
      int* dangerous = new int;  // What if there is no memory?
      if (dangerous == 0)                 // Now we are safe
            exit(1);

      set_new_handler(outOfMemory);
      int* safe = new int;
                              // This is safe, if there is no memory
                              // OutOfMemory will be called for us
}
```

Typical implementation of new

```
void * operator new(size_t size) {
        void *p;

        if(size == 0)               /* allow new(0)   */
                size = 2;

        while (1) {
                p = __cp_malloc(size);
                if (p != NULL || _new_handler == NULL)
                        break;
                (*_new_handler)();
        }
        return p;
}
```