Functions in C++

Chapter 3

Objectives

- **Use function prototypes in your code.**
- **Make use of automatic conversion of parameters in function calls when there is a prototype.**
- **Use inline functions.**
- **Use default arguments.**
- **Define "overloaded" and explain the benefits of overloading.**
- **Describe the standard C/C++ call by value mechanism for passing parameters in functions calls.**
- **Gain experience through code walk-throughs and lab exercises.**

  - The example programs are in the **chapter directory.**
  - Labs located in Labs/Lab3

Function Prototypes in C++

- **Function prototypes:**
  - Pioneered in C++
  - Part of ANSI C
  - Mandatory in C++

- *type function*(*type1*, . . . , *typeN*);

Prototype Usage

- **Each type is a primitive type, a type expression or user defined type.**
  - Use **void** if no function return.
  - Can optionally use **void** for an empty argument list.
  - Optionally can have variable names after types in argument list to improve readability.

Strong Type Checking

- **Argument list and return type of every function call are type checked during compilation.**
- **Number of arguments must agree.**
- **Types of arguments and return value must agree either through an exact match or through an implicit type conversion.**

Prototype	Call
void foo(int);	x = foo(7); // illegal
void foo(int);	foo(7); // legal
void foo(int);	foo(7, 12); // illegal
void foo(int);	foo(3.14) ; // legal
void foo(int);	foo(“Hi"); // illegal

Conversion of Parameters

- **Use of function prototypes causes parameters to be converted automatically in function calls:**

```
int num_digits(long x);

short a = 6789;

int n;

n = num_digits(a); // a is converted to long
```

- **Classes can implement conversion operations, which behave the same way as conversion of built-in types.**
  - *Later we will see how constructors can be used to accomplish type conversion.*

Inline Keyword

- **Use the *inline* keyword in definition of function to cause it to be expanded inline, saving function call overhead at run-time (but may use more space):**

```
inline float cuberoot(float x)
{
    return  exp(log(x) / 3.0);
}
```

Inline Usage

- **Inline functions of interest to more than a single file must be placed in a header file.**
- **Inline member functions of a class do not need the *inline* keyword.**
- **Type checking is done (unlike macros).**
- **Inline functions can be overloaded.**
- **Inline is a *hint* to the compiler (complex functions, e.g. involving recursion, may not be inlined).**
- **Within each file that an inline function is used but cannot be expanded, a static definition of the function is generated.**
  - This can result in multiple static instances being defined within a single executable.

Inline Code Example

- Open and examine the file IntStack.h in folder InlineStack.
- The specification and implementation are now contained in a single file.

Default Arguments

- **A formal parameter can be given a default argument.**

```
int power(int x,int p = 2); // this would normally be in the public header file
int power(int x, int p) {
       int prod = 1;
       for (int i = 1; i <= p; ++i) {
         prod  *= x;
       }
       return  prod;
    }
power(5, 3);            // answer is 125
power(5);               // answer is 25
```

Function Overloading

- **One name can be used for several similar functions.**
- **Functions must have different number of parameters:**

```
void foo(int);

void foo(int, int);
```

- . . . or different types of parameters:

```
void foo(int);

void foo(float);
```

- Function selection is based on matching types of parameters (the signature of the function).

Argument Matching

- **A call to an overloaded function is resolved to a particular instance through argument matching.**
- **Argument matching is attempted in the following order:**
  - An exact match.
  - A match through *promotion*.
  - A match through *standard conversion*.
  - A match through *user-defined conversion*.

Argument Matching through Promotion

- **Promotion maps a data type into a more inclusive data type.**
  - **char** promotes to **int**
  - **int** promotes to **long**

- **An example of type conversion can be found in folder** Max.

Match through Type Conversion

- **Promotion is a special kind of type conversion, which involves "widening" of a data type, which will never lose information.**
- **Other standard conversions involve "narrowing", e.g.**
  - int e = 2.71828;   // e is 2

User-Defined Type Conversions

- **Classes can incorporate *user-defined* type conversions, which can also be used to resolve overloaded function calls.**
  - Constructors can provide one means of type conversion.
  - Later we will learn how to override cast operators to provide another means of type conversion.

Call By Value

- **In C and C++ the standard mechanism for passing parameters in function calls is call by value.**
- **A local copy is made of each parameter:**

```
void increment(int x){
  ++x;
}
a = 2;
increment(a); // a is still 2; only the copy was changed
```

- **Review sample application in folder CallByValue.**

Ramifications

- **Call by value has many ramifications in C and C++.**
  - In C++ when an argument is an **object**, the compiler will create a new temporary object as part of the function call operation.
  - In C you must use **pointers** when you want to get a changed value of a parameter back to the calling program.
  - In C++ there is an alternative parameter passing mechanism, **call by reference**, which we will discuss in a later chapter.

Summary

- **C++ introduced function prototypes, which have been incorporated into ANSI C.**
- **Prototypes can be used to generate code to automatically convert types of parameters, but you must be careful in cases of a variable number of arguments.**
- **Inline functions have speed of macros and type safety of ordinary functions.**
- **Default arguments can be used to avoid passing a frequently occurring value on each function invocation.**
- **One name can be used for several functions. Such a name is said to be *overloaded*.**
- **A call to an overloaded function is resolved to a particular instance through *argument matching*.**
- **The standard parameter passing mechanism in C++ is *call by value*, which involves the compiler creating a copy of the arguments.**