C++ Notebook

An overview of the most important C++ features and concepts - not a complete reference.

by Lasse Rautiainen

Contents

Foreword

This notebook presents an overview of the most important C++ features and concepts - this is not a complete reference. I have gathered all this knowledge while I was refreshing my knowledge of the language (see References at end). Primarily this is a notebook for me but I hope that it will also help other people to find answers to their questions.

"Axioms in philosophy are not axioms until they are proved upon our pulses: we read fine things but never feel them to the full until we have gone the same steps as the author.", John Keats (1795-1821), Letter to J.H. Reynolds, May 3, 1818.

Lasse Rautiainen, 9.9.2003

1 Introduction

C++ is a general-purpose programming language based upon the ANSI standard C language. C language is one of the most liked and widely used professional programming languages in the world. C++ adds extensions to C that support data abstraction, object-oriented programming, and genereric programming. The C++ extensions were first invented by Bjarne Stroustrup in 1980 at Bell Laboratories in Murray Hill, New Jersey.

The reason for invention was complexity of huge C programs. C programs with hundreds of thousands lines of code are too complex and very difficult to grasp as a totality. The essence of C++ is to allow the programmer to comprehend and manage larger, more complex programs. Althought C++ was initially designed to aid in the management of very large programs, it is in no way limited to this use. The object-oriented attributes of C++ can be effectively applied to any programming task for example such as editors, databases, personal file systems, communication programs, and games.

Did You Know?

- C++ was initially called "C with Classes". However, in 1983 the name was changed to C++.
- Some of C++'s object-oriented features were inspired by another object-oriented language called Simulta67.
- July 1998 - NCITS (National Committee for Information Technology Standards) announced the approval of Programming Language C++ both Nationally and Internationally.

1.1 Object-Oriented Programming

When Stroustrup was developing C++, he managed to maintain the original spirit of C, including its efficiency, flexibility and underlying philosophy that the programmer, not the language, is in charge, while at the same time adding support for Object-Oriented Programming (OOP). OOP has taken the best ideas of structured programming and combined them with several powerful new concepts that encourage programmer to approach the task of programming in a new way. In general a problem is decomposed into subgroups of related parts that take into account both code and data related to each group. These subgroups are organized into a hierarchical structure and finally translated into self-contained units called objects.

All OOP languages have three things in common: objects, inheritance, and polymorphism.

1.1.1 Objects

Object is the most important feature of an object-oriented language. An object is a logical entity that contains both data and code that manipulates that data. An object provides a level of protection against some other, unrelated part of the program accidentally modifying or incorrectly using the private parts of the object - this is often referred to as *encapsulation*.

1.1.2 Inheritance

Inheritance is the process by which one object can acquire the properties of another object. This is important because it supports the concept of classification. Most knowledge is made manageable by hierarchical classifications. For example, a green raw banana is part of the classification *banana*, which in turn is part of the *fruit* class, which is under the larger class *food*. Without the use of

classifications, each object would have to define explicity all of its characteristics. However, through the use of classifications, an object need only define those qualities that make it unique within its class. It is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

1.1.3 Polymorphism

Polymorphism is characterized by the phrase "one interface, multiple methods." This means that one name can be used for several related but slightly different purposes. In essence, polymorphism allows one interface to be used with a general class of actions. The specific action selected is determined by the type of data involved.

In C++, run-time and compile-time polymorphism are supported.

1.1.4 Terminology

Along with each programming revolution comes a new set of terminology. There are some new OOP concepts, but many have a simple analog in pre-OOP practice.

Table 1.1: OOP Terminology

OOP Term	Definition
method	Same as function, but the typical OO notation is used for the call, ie, f(x,y) is written x.f(y) where x is an object of class that contains this f method.
send a message	Call a function (method).
instantiate	Allocate a class/struct object (ie, instance) with new.
class	A struct with both data and functions.
object	Memory allocated to a class/struct. Often allocated with new.
member	A field or function is a member of a class if it's defined in that class.
constructor	Function-like code that initializes new objects (structs) when they instantiated (allocated with new).
destructor	Function-like code that is called when an object is deleted to free any resources (eg, memory) that is has pointers to.
inheritance	Defining a class (child) in terms of another class (parent). All of the public members of the public class are available in the child class.
polymorphism	Defining functions with the same name, but different parameters.
overload	A function is overloaded if there is more than one definition. See polymorphism.
override	Redefine a function from a parent class in a child class.
subclass	Same as child, derived, or inherited class.
attribute	Same as data member or member field.

2 Basic Information

This chapter provides the basic information of C++ programming. First, the structure of a C++ program is introduced. Second, preprocessor directives are discussed. Finally, the common mystery around namespaces is exposed.

2.1 Structure of a C++ Program

A program is a set of human-readable instructions for the computer to carry out. The human-readable instructions (or statements) are turned into a computer executable program by a computer.

The first program that most programming apprentices write for the first time, is to print on screen the "Hello World!" sentence. It is one of the simpler programs that can be written in C++, but it already includes the basic components that every C++ program has.

The most books introduce the "Hello World!" example more or less like this:

```
#include <iostream>

int main() {
  std::cout << "Hello world in ANSI-C++\n";

  return 0;
}
```

This could be a bit more interesting. Here is a basic C++ framework that does exactly the same but is a far more close to real C++ programming.

File: mybase.h

```
/* ------------------------------------------------------
   This file contains the declaration of the class "MyBase"
   ------------------------------------------------------ */
#ifndef MYBASE_H // check for prior inclusion
#define MYBASE_H

#include "includes.h"

//  global functions, variables

class MyBase { // class definition
public:
  // public variables
  MyBase(void);          // constructor(s)
  virtual ~MyBase(void); // destructor
  virtual ShowMe(void);  // public functions
protected:
  // protected variables and function
private:
  // private variables and function
};

#endif
```

File: includes.h

```
/* ------------------------------------------------------
   This file contains the declaration of standard variables,
   and functions
   ------------------------------------------------------ */
#ifndef CPP_INCL_H   // check for prior inclusion
#define CPP_INCL_H

#include <iostream>  // C++ I/O functions
using namespace std; // make str names available without std:: prefix

#endif
```

File: mybase.cpp

```
/* ------------------------------------------------------
   This file contains the implementation of "MyBase"
   ------------------------------------------------------ */
#include "mybase.h" // includes

MyBase::MyBase(void) { // constructor
  cout << "Constructing..." << endl;
}

MyBase::~MyBase(void) { // destructor
  cout << "Destructing..." << endl;
}

MyBase::ShowMe(void) {
  cout << "Hello World!" << endl;
}
```

File: main.cpp

```
/* ------------------------------------------------------
   This file contains the main() program to test
   the implementation of the class "MyBase"
   ------------------------------------------------------ */
#include "mybase.h"

int main() {
  MyBase a_base ;

  a_base.ShowMe();

  return 0;
}
```

The files, listed above, encompass a basic C++ framework. The file mybase.h contains the declarations for the class (see 8 Classes) MyBase. The file mybase.cpp contains the definitions of those functions (see 7 Functions) and variables (see 3 Expressions) declared in mybase.h. The file main.cpp contains the main function (required in all C/C++ programs) within which an instance of MyBase is created and it's member function ShowMe is called. The file includes.h is a convenience file, the contents of which could be entered directly into mybase.h. However, if two or more classes are created requiring more or less the same **#include** statements, it is easier to maintain them in one file and reference it when needed.

2.1.1 Comments

Comments are pieces of source code discarded from the code by the compiler. They do nothing. Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code.

C++ supports two ways to insert comments:

```
// line comment
/* block comment */
```

The first of them - the line comment, discards everything from where the pair of slash signs (//) is found up to the end of that same line. The second one, the block comment, discards everything between the /* characters and the next appearance of the */ characters, with the possibility to include several lines.

2.1.2 Includes

Sentences that begin with a pound sign (#) are directives for the preprocessor. They are not executable code lines but indications for the compiler. In the example the sentence **#include <iostream>** tells the compiler's preprocessor to include the iostream standard header file. This specific file includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is used later in the program.

Preprocessor directives are examined more deeply in the chapter 2.2 Preprocessor Directives.

2.1.3 Namespace

The concept of namespace is described in the chapter 2.3 Namespaces.

2.1.4 Main Function

The main function is the point by where all C++ programs begin their execution. It is independent from whether it is at the beginning, at the end or by the middle of the code - its content is always the first to be executed when a program starts. In addition, for that same reason, it is essential that all C++ programs have a main function.

2.1.5 Code Blocks

Code block is a set of statements that fall between open and closing brackets: "{" "}". The code that lies between the brackets constitutes the code block. The code within the block can consist of any number of statements. Each statement must end with a semicolon: ";".

2.1.6 Functions

A C++ program consists of a series of one or more functions. A *function* is a program (or code) that performs a task. Formally, a function is a subprogram called from within an expression that has a single value that is computed and returned to the main program.

2.2 Preprocessor Directives

Preprocessor directives are orders that are included within the code of the programs that are not instructions for the program itself but for the preprocessor. The preprocessor is executed automatically by the compiler when a program is compiled in C++ and is the one in charge to make the first verifications and digestions of the program's code.

Table 2.1: The preprocessor directives

Directive	Description
#include	When the preprocessor finds an #include directive it replaces it by the whole content of the specified file.
#define	Serves defined constantants or macros.
#undef	Fulfills the inverse functionality than #define. It eliminates from the list of defined constants the one that has the name passed as parameter to #undef.
#ifdef	Allows that a section of a program is compiled only if the defined constant that is specified as parameter has been defined, independently of its value.
#ifndef	The code between the #ifndef directive and the #endif directive is only compiled if the constant name that is specified has not been defined previously.
#if, #else and #elif (elif = else if)	Directives serve for that the portion of code that follows is compiled only if the specified condition is met. The condition can only serve to evaluate constant expressions.
#line	When we compile a program and there happens any errors during the compiling process, the compiler shows the error that have happened preceded by the name of the file and the line within the file where it has taken place.
#error	This directive aborts the compilation process when it is found returning the error that is specified as parameter.
#pragma	This directive is used to specify diverse options to the compiler. These options are specific for the platform and the compiler you use. Consult the manual or the reference of your compiler for more information on the possible parameters that you can define with #pragma.

All these directives must be specified in a single line of code and they do not have to include an ending semicolon (;).

Examples

```
#include "file" // looks from the same directory
#include <file> // looks from the default directories

#define MAX_WIDTH 100
#define getmax(a,b) a > b ? a : b

#undef MAX_WIDTH

#ifdef MAX_WIDTH
char str[MAX_WIDTH];
#endif

#ifndef MAX_WIDTH
#define MAX_WIDTH 100
#endif

#if MAX_WIDTH > 200
#undef MAX_WIDTH
#define MAX_WIDTH 200
#elif MAX_WIDTH < 50
#undef MAX_WIDTH
#define MAX_WIDTH 50
#else
#undef MAX_WIDTH
#define MAX_WIDTH 100
#endif

#line 1 "assigning variable"
int a?;
// This code will generate an error that will be shown as error
// in file "assigning variable", line 1.

#ifndef __cplusplus
```

```
#error A C++ compiler is required
#endif
```

2.3 Namespaces

Namespaces allow to group a set of global classes, objects and/or functions under a name. They serve to split the global scope in sub-scopes known as namespaces. The general form of a namespace is

```
namespace identifier {
  namespace-body
}
```

Where identifier is any valid identifier and namespace-body is the set of classes, objects and functions that are included within the namespace.

Example

```
#include <iostream>

namespace first {
  int var = 1;
}

namespace second {
  double var = 2;
}

int main () {
  std::cout << first::var << endl;
  std::cout << second::var << endl;
  return 0;
}
```

2.3.1 Using Namespace

The using directive followed by namespace serves to associate the present nesting level with a certain namespace so that the objects and functions of that namespace can be accesible directly as if they were defined in the global scope.

Example

```
#include <iostream>

namespace first {
  int var = 1;
}

namespace second {
  double var = 2;
}

int main () {
  {
    using namespace first;
    std::cout << var << endl;
  }
  {
    using namespace second;
    std::cout << var << endl;
  }
  return 0;
}
```

2.3.2 Namespace std

Almost all compilers, even those complying with ANSI standard, allow the use of the traditional header files (like iostream.h, stdlib.h, etc). Nevertheless, the ANSI standard has completely redesigned this libraries taking advantage of the templates feature and following the rule to declare all the functions and variables under the namespace std.

The standard has specified new names for these "header" files, basically using the same name for C++ specific files, but without the ending .h. For example, iostream.h becomes iostream.

Example

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

int main () {
  cout << "Hello world!";
  return 0;
}
```

3 Expressions

The expressions are the most fundamental elements of the C++ language. Expressions are formed from the atomic elements of C++: data and operators. C++ supports a number of different types of data. Data may be represented either by variables or by constants.

3.1 Data Types

C++ has a set of basic data types: boolean (**bool**), character (such as **char**), integer (such as **int**), and floating-point (such as **double** and **float**). With the exception of type **void**, the base types can be modified by using a signed, unsigned, long, or short *modifier*. The type **void** has two uses. It is used either to specify that a function does not return a value or as the base type for pointers to objects of unknown type. In addition, a user can define enumeration types for representing specific sets of values (**enum**).

From the types mentioned above, other types can be constucted: pointer (such as **int***), array (such as **char[]**), reference (such as **double&**), and data structures and classes.

3.1.1 Boolean

A Boolean can have one of the two values *true* or *false*. By definition, *true* has the value 1 when converted to an integer and *false* has the value 0.

Table 3.1: Boolean Type

Type	Range	Size in Bits
bool	0,1	1

Examples

```
bool b = true;

int i = false; // int(false) is 0, so i becomes 0

bool b = 4; // bool(4) is true, so b becomes true

bool smaller(int i, int j) { return i<j; }
```

3.1.2 Character

A variable of type **char** can hold a character of the implementation's character set. A type **wchar_t** is provided to hold characters of a larger character set such as Unicode. The Unicode standard specify the Universal Character Set (UCS), a character set that allows character units to be processed for all languages with the same set of rules.

Table 3.2: Character Types

Type	Range	Size in Bits
char	-127 to 127	8
signed char	-127 to 127	8

Type	Range	Size in Bits
unsigned char	0 to 255	8

A few characters have standard names that use the backslash (\) as an escape character.

Table 3.3: Escape Characters

Character	ASCII	Description
\n	NL (LF)	newline
\t	HT	horizontal tab
\v	VT	vertical tab
\b	BS	backspace
\r	CR	carriage return
\f	FF	form feed
\a	BEL	alert
\\	\	backslash
\?	?	question mark
\	'	single quote
\"	"	double quote
\ooo	ooo	octal number
\xhhh...	hhh	hex number

Examples

```
char ch = 'a';
wchar_t wch = L'L';
```

3.1.3 Integer

Even the **bool** and **char** data types are already introduced, those types are actually binary integers and could be presented here.

Table 3.4: Integer Types

Type	Range	Size in Bits
int	-32 767 to 32 767	16
signed int	-32 767 to 32 767	16
unsigned int	0 to 65 535	16
short int	-32 767 to 32 767	16
signed short int	-32 767 to 32 767	16
unsigned short int	0 to 65 535	16
long int	-2 147 483 647 to 2 147 483 647	32
signed long int	-2 147 483 647 to 2 147 483 647	32
unsigned long int	0 to 4 294 967 295	32

Integer literals come in four guises: decimal, octal, hexadecimal, and character literals.

Examples

```
int i = 123;         // decimal

unsigned int i = 1U; // the suffix U or L can be used to
long int i = 4L;     // write explicitly unsigned literals

int i = 0123;        // octal

int i = 0x0;         // hexadecimal

int i = 'A';         // character literal
while (i <= 'Z') printf("\n%c", i++);
```

3.1.4 Floating-Point

The floating-point types represent floating-point numbers. Floating-point types come in three sizes: **float** (single-precision), **double** (double-precision), and **long double** (extended precision).

Table 3.5: Floating-Point Types

Type	Range	Size in Bits
float	1e-37 to 1e+37, 6 digits of precision	32
double	*, 10 digits of precision	64
long double	*, 10 digits of precision	128

* The **float** and **double** magnitudes will depend upon the method used to represent the floating-point numbers.

Examples

```
float f = 1.23;

float f = 2.0f;

double d = .123;

long double d = 1.2e10;
```

3.1.5 Void

Examples

```
void f(); // function does not return a value

void* pa; // pointer to object of unknown type
```

3.1.6 Enumerations

An *enumeration* is a type that can hold a set of values specified by the user. The enum declaration creates a new integer type. By convention the first letter of an enum type should be in uppercase. The list of values follows, where the first name is assigned zero, the second 1, etc. It is also possible to control the values that are assigned to each enum constant.

Examples

```
enum Align { TOP=1, RIGHT, BOTTOM, LEFT }; // counting starts from 1
enum Color { RED=2, BLUE=4 };
...
```

```
Align a;
Color b;
...
a = BOTTOM;
b = RIGHT;    // bad, but legal in C++
b = 10;       // even this is legal
```

3.1.7 Casts

Casts can be used to make type conversion clear, and especially when "narrowing" range of a value.

Examples

```
float f = 2.0f;
int i = int(f); // use function form only with simple type names
int i = (int)f;
```

3.1.8 Composite

In addition to the simple data types (int, char, double, ...) there are composite data types which combine more than one data element. Arrays are used to store many data elements of the same type. Structs (also called records) group elements which don't need to all be the same type. Classes are like structs where the members are private by default. Classes are used for object-oriented programming where functions are defined in addition to the data members. If there are only data members and no functions, it is common to use structs instead of classes.

Did You Know?

- The size and range of data types vary with each prosessor type and with the implementation of the compiler. The ANSI C standard stipulates only the minimal *range* of each data type. Values outside the range may be handled differently between C and C++ implementations.

3.2 Variables

A *variable* is a named location in memory that is used to hold a value which may be modified by the program. All C++ variables must be declared before they are used. The general form of a declaration is

```
type variable = initial;
```

Here, *type* must be a valid C++ data type (see 3.1 Data Types), and *variable* may consist of one or more identifier names with comma separators. Initial is the initial value (optional).

In C/C++ the names of variables, functions, labels, and various other user-defined objects are called *identifiers*.An identifier can vary from one to several characters. The first character must be a letter or an underscore; subsequent characters must be letter, numbers, or an underscore.

An identifier may not be the same as a C or C++ keyword (see Appendix: C/C++ Keywords), and it should not have the same name as a function that you wrote or that is in the C or C++ library. In C++, there is no limit to the length of an identifier and all characters are significant.

Examples

```
int i, j, k;
```

```
char ch;

double current_balance;
```

The name of a variable has nothing to do with its type. It is recommended to use prefixes which will indicate the type and maintain a consistent naming style. For example, iPixel (**int**), chName (**char**) etc. Variable names should start with a lowercase letter. Second words in a name should start with an uppercase letter.

3.2.1 Variable Locations

Variables can be located inside functions (local variables), in the definition of function parameters (formal parameters), and outside all functions (global variables).

In C++, local variables can be defined at any point in a program.

Example

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

int i;               // global variable

void f(int i, int j, int k); // function prototype

int main()
{
  i = 1;             // local variable

  int j;
  j = 2;
    {
      int j = 3;     // hides first local j
      int i = 2;     // hides global i
      ::i = 4;       // assign to global i
    }
  int k;             // this is an error in C, but not in C++
  k = 3;

  f(i, j, k);
  return 0;
}

void f(int i, int j, int k) // formal parameters
{
  cout << i << " " << j << " " << k <<
endl;
}
```

3.2.2 Access Type Modifiers

There are two type modifiers that may be used to control the ways in which variables may be accessed or modified: **const** and **volatile**.

Variables of type **const** may not be changed by your program. However, initial value can be given. Very important use of variables of type **const** is to protect arguments to a function from being modified by that function. That is, when a pointer is passed to a function, it is possible for that function to modify the actual variable pointed to by the pointer.

The keyword **const** is also used in front of a declaration which has an initial value. Constant names

should be in uppercase characters. If the name has multiple words, they should be separated by underscore.

A **volatile** modifier is a hint to a compiler that an object may change its value in ways not specified by the language so that aggressive optimizations must be avoided.

Examples

```
const MAX_SESSIONS = 500; // Max number of sessions.

const int i = 1;

void f(const char* p)
{
  ... // pointer p cannot be modified here
}

const volatile clock; // two reads of clock give different results
```

3.2.3 Strorage Class Type Modifiers

The four storage class modifiers are: **auto**, **extern**, **register**, and **static**. Storage class type modifiers tell the compiler how to store the subsequent variable. The storage specifier precedes the rest of the variable declaration.

```
storage_specifier type variable;
```

Specifier **auto** is defined within a block and/or the nested blocks; when the same variable name is re-defined inside of a nested block, the previously named variable becomes "hidden." When exiting the block, the value of the variable is lost. When entering the block, the value of the variable will get re-initialized.

Specifier **extern** is defined for the entire duration of the program execution and all functions may access and modify the variable. The variable is initialized once.

Specifier **register** is same as auto class; this is only a recommendation to the C/C++ compiler on an attempt to increase execution speed; This will be defaulted to auto class whenever the compiler cannot allocate an appropriate physical register; typically used with frequently accessed variables.

Specifier **static** is defined within a block only; will become "hidden" when exiting the block. When re-entering the block, the variable is awaken and retains the previous value. The variable is initialized once.

Example

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

void printValue(void);   // extern
int i = 1;               // extern

void main(void)
{
  int j = 5; // auto class
  extern int i;
  register int k;
  for (k = 0; k < 2; k++) {
    i += j;
    printValue();
```

```
  }
}

void printValue(void)
{
  static int Times = 0;
  Times++;
  cout << "This function has been called " << Times << "
times." <<
    endl;
  cout << i << endl;
}
```

Did You Know?

- A variable of type **const** can be modified by something outside your program; for example, a hardware device may set its value.

3.3 Operators

C++ is rich in built-in operators. It places significantly more importance on operators than do most other computer languages. C++ defines several classes of operators: assigment, arithmetic, comparison, logical, conditional, bitwise, and reference. In addition, C++ has some special operators for particular tasks and operators can be even overloaded.

Because there is a lot of operators, there must be a specific execution order. C++ operators by precedence are described in the Appendix: Precedence. In practise, it is sufficient to remember only unary operators, *, /, %, +, -, comparisons, &&, ||, and = assignments. And use parenthesis for all others.

3.3.1 Assignment

The assignation operator serves to assign a value to a variable. Assignment operators are provided for all binary operators except && and ||.

Table 3.6: Assignment Operators

Operator	Description
=	simple assignment
+=	add and assign
-=	subtract and assign
*=	multiply and assign
/=	divide and assign
%=	modulo and assign
&=	AND and assign
|=	inclusive OR and assign
^=	exclusive OR and assign
<<=	shift left and assign
>>=	shift right and assign

Examples

```
int i, j; // i:?  j:?
i = 1;    // i:1 j:?
j = 2;    // i:1 j:2
```

```
i = j;    // i:2 j:2
j = 3;    // i:2 j:3
```

3.3.2 Arithmetic

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The module, specified with the percentage sign (%), is the operation that gives the rest of a division of two integer values.

The increase operator (++) and the decrease operator (--) are an example of saving when writing code. They increase or reduce by 1 the value stored in a variable. They are equivalent to +=1 and to -=1, respectively. Thus, **a++;** is equivalent with **a+=1;** is equivalent with **a=a+1;**.

Table 3.7: Arithmetic Operators

Operator	Description
opr1 + opr2	addition
opr1 - opr2	subtraction
opr1 * opr2	multiplication
opr1 / opr2	division
opr1 % opr2	remainder (modulo) after dividing opr1 by opr2
++opr	add 1 to operand before using the value
--opr	subtract 1 from operand before using the value
opr++	add 1 to operand after using the value
opr--	subtract 1 from operand after using the value

The result of arithmetic operators is double if either operand is double, else float if either operand is float, else long if either operand is long, else int.

Examples

```
int i;      // i:?
i = 1;      // i:1
i++;        // i:2
i = 11 % 3; // i:2
i = 13 / 4; // i:3
```

3.3.3 Comparison

In order to evaluate a comparison between two expressions we can use the Relational operators. ANSI-C++ standard specifies that the result of a relational operation is a **bool** value that can only be **true** or **false**.

Table 3.8: Comparison Operators

Operator	Description
==	Equal
!=	Different
>	Greater than
<	Less than
>=	Greater or equal than
<=	Less or equal than

Examples

```
(1 == 2)    // false
(1 != 2)    // true

int i = 1;
int j = 4;
(i <= j)    // true
(i > j)     // false
```

Be aware. Operator = (one equal sign) is not the same as operator == (two equal signs), the first is an assignation operator (assigns the right side of the expression to the variable in the left) and the other (==) is a relational operator of equality that compares whether both expressions in the two sides of the operator are equal to each other.

3.3.4 Logical

Logic operators && and || are used when evaluating two expressions to obtain a single result. They correspond with boolean logic operations AND and OR respectively. Operator ! is equivalent to boolean operation NOT, it has only one operand, located at its right.

Table 3.9: Logical Operators

Operator	Description
opr1 && opr2	Conditional "and". true if both operands are true, otherwise false.
opr1 || opr2	Conditional "or". true if either operand is true, otherwise false.
!opr	true if opr is false, false if opr is true

Examples

```
if (character >= 'A' && character <= 'Z')
  cout << "Uppercase character" << endl;

if (income >= 100000.00 || cash >= 1000000.00)
  cout << "Why you want to loan money?" << endl;
```

3.3.5 Conditional

The conditional operator evaluates an expression and returns a different value according to the evaluated expression, depending on whether it is true or false.

Table 3.10: Conditional Operators

Operator	Description
cond ? res1 : res2	if cond is true, the value is res1, else res2. Results must be the same type.

Examples

```
1==2 ? 1 : 2 // returns 2 since 1 is not equal to 2.
```

3.3.6 Bitwise

Bitwise operators modify the variables considering the bits that represent the value they store, that means, their binary representation.

Table 3.11: Bitwise Operators

Operator	Description
opr1 & opr2	bitwise AND
opr1 | opr2	bitwise inclusice OR
opr1 ^ opr2	bitwise exclusive OR (XOR)
~ opr	complement
opr1 << opr2	shift right
opr1 >> opr2	shift left

Examples

```
char x = 7;   // value of x: 7 = (00000111)
x = x << 1;     // value of x: 14 = (00001110)

// a simple encryption function.
char encrypt(char ch)
{
  return (~ch); // complement
}
```

3.3.7 Reference

A *reference* is an alternative name for an object. The main use of references is for specifying arguments and return values for functions in general and for overloaded operators in particular.

Table 3.12: Reference Operators

Operator	Description
object .	member member selection
pointer ->	member member selection
pointer [expr]	subscripting
& opr	address of operand
* expr	deference
object .*	pointer-to-member member selection
pointer ->*	pointer-to-member member selection
std::cout	scope resolution

Examples

```
void strcpy(char* a, char* b) {
  while (*a++ = *b++) ; // assignment, not comparison!
}

void incr(int* p) { (*p)++; }
...
int  i = 1;
int& r = i; // r and i refer to the same int
int  j = r; // x = 1
r = 2;      // i = 2

incr(&i);   // i = 3
...
struct name {
  char* firstName;
  char* lastName;
};
```

```
void print_name(name* p)
{
  cout << p->firstName << '\n'
       << p->lastName << endl;
}
...
name nm;
char* name;     // a pointer to a char
name = "Lasse"; // assigns the address of the first char

strcpy(nm.firstName, name);
nm.lastName = "Rautiainen";
print_name(nm);
```

3.3.8 I/O

There are two operators defined in <iostream> library which can be presented here.

Table 3.13: I/O Operators

Operator	Description
cout << opr	output "insertion"
cin >> opr	input "extraction"

3.3.9 Miscellaneous

The standard library string provides an operator for string concatenation.

The sizeof operator can be used to find out how much memory a type uses. Technically it is operator even it is usually written as function.

Table 3.14: Miscellaneous Operators

Operator	Description
opr1 + opr2	string concatenation"
sizeof opr	memory size of opr in bytes

3.3.10 Scope Resolution

The :: operator is used to link a class name with a member name in order to tell the compiler what class the member is part of. It can also allow access to a name in an enclosing scope that is "hidden" by a local declaration of the same name.

Examples

```
int i;      // global i

void f()
{
  int i;    // local i

  i = 10;   // uses local i
  ::i = 10; // refers to global i
}
```

4 Statements

A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, C++ provides program statements that serve to specify what and how has to perform our program.

Expression statements are simply statements made up of a valid C++ expression. Block statements are simply bocks of code. (A block begins with **{** and ends with **}**.) The other statements are categorized here into three groups: selection, iteration, and jump. The selection (conditional) statements are **if** and **switch**. The iteration (loop) statements are **while**, **do-while**, and **for**. The jump statements are **break**, **continue**, **goto**, and **return**.

4.1 Selection

Selection statements choose one of several flows of control.

```
if ( expression ) statement

if ( expression ) statement else statement

switch ( expression ) {
  case constant1:
    statement
    break;
  case constant2:
    statement
    break;
  ...
  default:
    statement
}
```

4.1.1 If

If the *if* expression evaluates to true (anything other than zero), the statement or block that forms the target of the if is executed; otherwise, if it exists, the statement or block that is the target of the else is executed. Only the code associated with the if or the code associated with the else will execute - never both.

Examples

```
if (a > b)
  c = a;
else
  c = b;

// following statement is alternative to replace if-else
// statements of the general form (above).
c = a > b ? a : b;

// it can be handy in some situations
int imouse = 0;
cout << "How many mice do you have?"
cin << imouse;
cout << "You have " << imouse <<
  (imouse == 1 ? " mouse" : " mice") << endl;
```

```
// nested ifs are common in programming
if (i) {
  if (j) statement 1;
  if (k) statement 2; // this if
  else   statement 3; // is associated with this else
}
else statement 4;     // associated with if (i)
```

4.1.2 Switch

The *switch* statement causes control to be transferred to one of several statements depending on the value of an expression. When the switch statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case label. If no case constant matches the expression, and if there is a default label, control passes to the statement labeled by the default label. If no case matches and if there is no default then none of the statements in the switch is executed.

Technically, the **break** statements inside the **switch** statement are optional. They terminate the statement sequence associated with each constant. If the **break** statement is omitted, execution continues on into the next **case**'s statements until either a **break** or the end of the **switch** is reached.

Example

```
cout << "Type a, b, or c: ";
cin >> ch;            // read user input

switch(ch)            // check the value of "ch"
{                     // the beginning of the switch statements scope
  case 'a': case 'A': // if ch equals 'a' or 'A', say so
    cout << "You typed a.";
    break;            // break out of the switch statement

  case 'b': case 'B':
 cout << "You typed b.";
 break;

  case 'c': case 'C':
 cout << "You typed c";
 break;

  default:            // if ch isn't a, b or c, say so
    cout << "You did not type a, b or c.";
}
```

4.2 Iteration

Iteration statements specify looping.

```
while ( expression ) statement;

do statement while ( expression );

for ( initialization; condition; increment ) statement;
```

4.2.1 While

The *while* loop iterates while the condition is true. When the condition becomes false, program control passes to the line after the loop code.

Example

```
// add spaces to the end of a string
void pad(char* s, int length)
{
  int l;
  l = strlen(s); // find out how long it is
  while (l < length) {
    s[l] = ' ';  // insert a space
    l++;
  }
  s[l] = '\0';   // strings need to be terminated in a null
}
```

4.2.2 Do-While

The *do-while* loop checks the condition at the bottom of the loop. This means that a do-while loop always executes at least once.

Example

```
#include <conio.h>  // include kbhit()
#include <iostream>

using namespace std;

int main()
{
  do {
    cout << "Still in the do...while loop" << endl;
  } while (!kbhit());
    // kbhit() is a function that returns TRUE if any
    // key on the keyboard has been pressed and FALSE otherwise
  return EXIT_SUCCESS; // The program terminated correctly
}
```

4.2.3 For

Most commonly, the *initialization* is an assignment statement used to set the loop control variable. The *condition* is a relational expression that determines when the loop will exit. The *increment* defines how the loop control variable will change each time the loop is repeated. These three major sections must be separated by semicolons. The *for* loop continues to execute as long as the condition is true. Once the condition becomes false, program execution resumes on the statement following the **for**.

Example

```
// function counts x power n
double power(double x, int n)
{
  double result = 1.0;
  if (n >= 0)
    for (int i = 0; i < n; i++)
      result *= x;
  else
    for (int i = 0; i < -n; i++)
      result /= x;
  return result;
}
```

4.3 Jump

Jump statements unconditionally transfer control.

```
break;

void exit(int return_code);

continue;

return expression;

goto identifier;
```

4.3.1 Break

The *break* statement has two uses. The first is to terminate a case in the switch statement. The second is to force immediate termination of a loop, bypassing the normal loop conditional test.

Example

```
int count = 1;

for (;;) { // infinite loop
  cout << count << endl;
  count++;
  if (count == 10) break;
}
```

4.3.2 Exit

A program can be break out by using the standard library function **exit()**. This function causes immediate termination of the entire program.

Example

```
int main()
{
  if (!color_card()) exit(1);
  play();

  return 0;
}
```

4.3.3 Continue

The *continue* statement is the complement to the break statement. Instead of forcing termination, continue forces the next iteration of the loop to take place, skipping any code in between.

Example

```
for (int space=0; *str; str++) {
  if (*str != ' ') continue;
  space++;
}
```

4.3.4 Return

The *return* statement is used to return from a function. A function can have many return statements.

However, function stops executing as soon as the first return is encountered. A function declared as void may not contain a return statement that specifies a value.

4.3.5 Goto

There is NOT a programming situation that requires use of the *goto* statement.

5 Arrays and Strings

An array is a collection of variables of the same type that are referenced by a common name. A specific element in an array is accessed by an index. Arrays may have one or several dimensions. The most common array is the array of char, which is simply an array of characters that is terminated by a null.

Strings in C++ are objects. C++ has had the potential for making arrays of characters much cleaner by providing a class library to implement strings.

5.1 Single-Dimension Arrays

Single-dimension arrays are essentially lists of information of the same type that are stored in contiguous memory locations in index order. All arrays have zero as the index of their first element. The general form of a single-dimension array is

```
type variable[size] = { initials };
```

Where type is the datatype of data to be stored, variable is the name of the array, size is an integer value, and initials are the values that will be stored (optional).

When an array is declared in C++, the compiler automatically creates an array pointer that points to the zero element of the array

There are essentially two ways to declare arrays in C++: statically and dynamically. The method you utilize in your program will depend on how you intend to use the array.

5.1.1 Static Arrays

Assigning a value results in a character array of finite length, established at compile time.

Examples

```
int numbers[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

char name[6];

name[0] = 'L';
name[1] = 'a';
name[2] = 's';
name[3] = 's';
name[4] = 'e';
name[5] = NULL; // every char array must end with a NULL character.

char name[6] = "Lasse";
name[0] = 'N';

char name[] = "Lasse"; // by not specifying a number of elements,
                       // this declaration will make name the
                       // appropriate length.

char* name = "Lasse";
name[0] = 'N'; // error: assigment to const; result is undefined
```

The problem with static arrays is that once you've declared the array size, you're stuck with it for the duration of the program. You have to know ahead of time how many elements you'll need storage for and reserve, in advance, that much memory. Memory for static arrays is carved out of the stack. An alternative to statically declaring arrays is to use dynamic array allocation.

5.1.2 Dynamic Arrays

Dynamic array allocation is actually a combination of pointers and dynamic memory allocation. Whereas static arrays are declared prior to runtime and are reserved in stack memory, dynamic arrays are created in the heap and released from the heap using the new[ ] and delete[ ] operators.

Examples

```
int *my_array;
my_array = new int[10];
delete[ ] my_array;
```

5.1.3 Array Parameters

If a function will be receiving a single-dimension array, the formal parameter can be declared in one of three ways: as a pointer, as a sized array, or as an unsized array. All three methods are indentical because each tells the compiler that a pointer is going to be received.

Examples

```
f(int *x)     // pointer
{
...
}

f(int x[100]) // sized array
{
...
}

f(int x[])    // unsized array
{
...
}
```

5.1.4 Array of Char Manipulation

C++ supports a wide range of array or char manipulation functions.

Table 5.1: Most common array of char manipulation functions.

Name	Function
strcpy(s1, s2)	Copies s2 into s1
strcat(s1, s2)	Concatenates s2 onto the end of s1
strlen(s1)	Returns the length of s1
strcmp(s1, s2)	Returns zero if s1 and s2 are the same; less than zero if s1 < s2; greater than zero if s1 > s2
strchr(s1, ch)	Returns a pointer to the first occurence of ch in s1
strstr(s1, s2)	Returns a pointer to the first occurence s2 in s1

Example

```
char name[] = "Lasse";

cout << "name[] length = "
     << strlen(name) << endl; // length is 5
```

Function **strlen()** returns the length of a string (char array) minus the NULL terminating character.

5.2 Multidimensional Arrays

Multidimensional arrays can be described as arrays of arrays. The general form of a multidimensional array is

```
type variable[size][size]...[size] = { initials };
```

Where type is the datatype of data to be stored, variable is the name of the array, size is an integer value, and initials are the values that will be stored (optional).

Example

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

#define WIDTH 5
#define HEIGHT 3

int data[HEIGHT][WIDTH];
int n,m;

int main ()
{
  for (n = 0; n < HEIGHT; n++)
    for (m = 0; m <WIDTH; m++)
    {
      data[n][m] = n+m;
    }
  return 0;
}
```

Multidimensional arrays can contain as many indices as needed, although it is rare to have to represent more than 3 dimensions. For example:

```
char data[100][100][100][100][100];
```

Assigns 10 billion chars which would consume about 10GB of RAM memory if it could be declared.

5.3 Strings

The C++ Standard Library provides a string class. To use this library, **#include <string>** must be added to the top of a program.

Unlike C-style strings, the internal representation of the string data is hidden by the string class. The data is set, accessed and manipulated using the methods of the string class. The programmer need not be concerned with how or where the string is stored. This is an example of encapsulation.

Example

```
#include <iostream>
#include <string>    // include for C++ standard string class
using namespace std;
```

```
void main()
{
  string szName = "Lasse";

  cout << "Length of szName = " << szName.length() <<
endl;
}
```

6 Pointers

Pointers are very important to C++. A pointer is a variable that holds a memory address. Most commonly, this address is the location of another variable in memory. If variable contains the address of another variable, the first variable is said to *point* to the second.

Pointers are so powerful that they are dangerous. They are dangerous because they can access any memory location and a small error in their use can have mysteriously bizarre results, often showing up only later in execution or when the program is run in a different environment. It is estimated that about 50% of the bugs in production software are due to pointer misuse.

A much safer and simpler use of memory addresses are *references*. Reference is the enhanced feature provided by C++. References are pointers which can't be manipulated with addition and subtraction. Removing this capability makes references much safer to use than pointers. In addition, they are automatically dereferenced so the programming notation is simpler and less error prone.

6.1 Expressions

In general, expressions involving pointers conform to the same rules as any other C++ expression. In this chapter a few special aspects of pointer expressions are examined.

6.1.1 Variables

If a variable is going to hold a pointer, it must be declared as such. The general form for declaring a pointer variable is

```
type* variable = &initial;
```

Where type defines that type of variables the pointer can point to, variable is the name of the pointer variable, and initial is the initial address (optional).

6.1.2 Operators

There are two pointer operators: * and &. The & is a unary operator that returns the memory address of its operand. The * is a unary operator that returns the value of the variable located at the address that follows.

Example

```
#include <iostream>

using namespace std;

int main()
{
  float f1, f2;
  int* p;

  p  = &f1;
  f2 = *p; // because p is integer pointer, only 2 bytes
           // will be transferred
  return 0;
```

```
}
```

6.1.3 Assigments

A pointer may be used on the assignment statements to assing its value to another pointer.

Example

```
int x;
int* p1;
int* p2;

p1 = &x;
p2 = p1; // address assigment
```

6.1.4 Arithmetic

There are two arithmetic operations that may be used on pointers: addition and subtraction. Each time a pointer is incremented, it points to the memory location of the next element of its base type. Each time it is decremented, it points to the location of the previous element.

Examples

```
int* p = 1000; // address assigment

p++;           // address is 1002; integers are 2 bytes long
p = p + 10;    // address is 1022
p--;           // address is 1020
```

6.1.5 Comparisons

The comparison operators (==, !=, <, >, <=, and =>) compare the addresses given by the two operands.

6.1.6 Pointers to Pointers

C++ allows the use of pointers that point to pointers, that these, on its turn, point to data. In order to do that we only need to add an asterisk (*) for each level of reference:

```
char a;
char* b;
char** c;
a = 'z';
b = &a;
c = &b;
```

6.1.7 Void Pointers

The type of pointer void is a special type of pointer. void pointers can point to any data type, from an integer value or a float to a string of characters. Its sole limitation is that the pointed data cannot be referenced directly (we can not use reference asterisk * operator on them), since its length is always undetermined, and for that reason we will always have to resort to type casting or assignations to turn our void pointer to a pointer of a concrete data type that we can refer. One of its utilities may be for passing generic parameters to a function:

```
#include <iostream>
```

```
using namespace std;

void increase (void* data, int type)
{
  switch (type)
  {
    case sizeof(char) : (*((char*)data))++; break;
    case sizeof(short): (*((short*)data))++; break;
    case sizeof(long) : (*((long*)data))++; break;
  }
}

int main ()
{
  char a = 5;
  short b = 9;
  long c = 12;
  increase (&a, sizeof(a));
  increase (&b, sizeof(b));
  increase (&c, sizeof(c));
  cout << (int) a << ", " << b << ", " << c;
  return 0;
}
```

6.2 Usage Targets

This chapter introduces a few practical usage targets of the pointers.

6.2.1 Arrays

The concept of array goes very bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, like a pointer is equivalent to the address of the first element that it points to, so in fact they are the same thing.

```
void f(char s[])
{
  for (char* p = s; *s != 0; s++)
    cout << *s << endl;
}

int numbers[5];
int* p;
p = numbers;  *p = 10;
p++;  *p = 20;
p = &numbers[2];  *p = 30;
p = numbers + 3;  *p = 40;
p = numbers;  *(p+4) = 50;
for (int n=0; n<5; n++)
  cout << numbers[n] << ", ";
```

6.2.2 Functions

C++ allows to operate with pointers to functions. The greater utility of that is for passing a function as a parameter to another function, since these cannot be passed dereferenced. In order to declare a pointer to a function we must declare it like the prototype of the function but enclosing between parenthesis () the name of the function and inserting a pointer asterisk (*) before.

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

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

```
int subtraction(int a, int b)
{ return (a-b); }

int(*minus)(int, int) = subtraction;

int operation(int x, int y, int (*functocall)(int, int))
{
  int g;
  g = (*functocall)(x,y);
  return (g);
}

int main ()
{
  int m,n;
  m = operation (7, 5, addition);
  n = operation (20, m, minus);
  cout << n;
  return 0;
}
```

6.2.3 Objects

Just as it is possible to have a pointer to many types of variables, it is possible to have pointers to objects. When accessing members of a class given a pointer to an object, the arrow (->) operator is used instead of the dot operator.

Example

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

class cl {
  int i;
public:
  cl() { i = 0; }
  cl(int j) { i = j; }
  int get_i() { return i; }
};

int main()
{
  cl object[3] = {10, 3, 7};
  cl* p;
  int i;

  p = ob; // start of array
  for (i = 0; i < 3; i++) {
    cout << p->get_i() << endl;
    p++;
  }
  return 0;
}
```

6.2.4 This

The "this" pointer addresses the object on which the method was called. Explicit use of the "this" pointer allows the concatenation of calls on an object. The "this" pointer is also useful if it is desired to use the same identifier for both a local variable within a method and for a class member. Also when implementing some methods, it is important to check for identity.

The compiler uses the "this" pointer to internally reference the data members of a particular object.

Example

```
class Point {
public:
  Point& setX(int x) {
    this->x = x;
    return *this;
  }
  Point& setY(int y) {
    this->y = y;
    return *this;
  }
  Point& doubleMe() {
    this->x *= 2;
    this->y *= 2;
    return *this;
  }
private:
  int x;
  int y;
};

...

Point a;
a.setX(1).setY(2).doubleMe(); // concatenation of calls
```

6.2.5 Derived Types

A pointer of a base class can be used as a pointer to any derived class. Althought a base pointer can be used to point to a derived object, it can access only the members of the derived type that were imported from the base.

Example

```
class base {
public:
  void set_i(int i)
  {
    this->i = i;
  }
private:
    int i;
};

class derived : public base {
public:
  void set_j(int j)
  {
    this->j = j;
  }
private:
    int j;
};

...

base* bp;
derived d;

bp = &d;
bp->set_i(1);             // correct
bp->set_j(2);             // incorrect
((derived*)bp)->set_j(3); // correct
```

6.2.6 Class Members

C++ allows to generate a pointer that points to a public member of a class, not to a specific instance of that member in an object. This is called a pointer to a class member, or a pointer to a member. To access a member of a class given a pointer to it, the special pointer-to-member operators .* and ->*. must be used.

Example

```
class base {
public:
  base(int i) {
    this->i = i;
  }
  void get_i() {
    return this->i;
  }
private:
  int i;
};

...

int (*func)();      // function member pointer
base ob(1);
base* p;

p = &ob;

func = &base:set_i; // get offset of get_i()

cout << (ob.*func)() << endl;
cout << (p->*func)() << endl;
```

6.3 References

Reference is a C++ feature that is related to the pointer. A reference is essentially an implicit pointer that acts as another name for an object.

6.3.1 Variables

Reference can be declared as a simply variable. This type of reference is called an independent reference. The general form of a declaration is

```
type &variable = reference;
```

Here, *type* must be a valid C++ data type (see 3.1 Data Types), and *variable* may consist of one or more identifier names with comma separators. Reference is the referenced identifier name (optional).

Examples

```
int i;
int &ref = i;

i = 1;
cout << a << " " << ref << endl;

ref = 2;
cout << a << " " << ref << endl;
```

```
ref++;
cout << a << " " << ref << endl;
```

6.3.2 Parameters

One important use for a reference is to allow to create functions that automatically use call-by-reference parameter passing rather than C++'s default call-by-value method.

Example

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

void swap(int &i, int &j);

int main() {
  int a = 1, b = 2;

  cout << a << " " << b << endl;
  swap(a, b);
  cout << a << " " << b << endl;

  return 0;
}

void swap(int &i, int &j) {
  int temp;

  temp = i;
  i = j;
  j = temp;
}
```

6.3.3 Return values

A function may return a reference. This allows a function to be used on the left side of an assignment statement.

Example

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

char &replace(int i); // return a reference
char szName[] = "Lasse";

int main() {
  replace(0) = 'N';

  cout << szName << endl;

  return 0;
}

char &replace(int i) {
  return szName[i];
}
```

6.3.4 Restrictions

There are a few restrictions that apply to references: Reference can not be another reference. Address

of reference can not be obtained. Array of references can not be created. A pointer to a reference is not allowed. A bit-field can not be referenced.

A reference variable must be initialized when it is declared unless it is a member of a class, a function parameter, or a return value.

7 Functions

Function is the basic element of all C++ programs. Splitting program into controlable parts is the idea which is the basis of all programming languages. Using functions programs can be structured in a more modular way, accessing to all the potential that structured programming in C++ can offer. Defining a function is the way to specify how an operation is to be done.

There is many reasons why the programs should be splitted. First, it eases reading and controlling process. Many of the nowadays applications consists millions lines of code and finding a bug from there might be a challenge without splitting. Second, functions can be reused. Standard library is a good example of function reuse. Third, splitting the program into several functions can reduce the needed amount of memory. By using appropriate functions reproducing of the code is avoided.

7.1 Definition

A function is a block of instructions that is executed when it is called from some other point of the program. The general form for declaring a function is

```
type name (arguments) statement
```

Where type is the type of data returned by the function, name is the name by which it will be possible to call the function, arguments are a comma-separated list of variable names and their associated types that will receive the values of the arguments when the function is called, and statement is the body of the function which can be a single instruction or a block of instructions. In the latter case it must be delimited by curly brackets {}.

All functions must be declared or prototyped before they are called. Each function should perform a single, well defined task, and its name should effectively express that task.

Example

```
// brick.h - header file
class Brick{
private:
  ...
public:
  int x; int y;
  ...
  bool move(int dx, int dy);     // function definition
  ...
};

// brick.cpp - source file
#include "brick.h"

...

bool Brick::move(int dx, int dy) // function implementation
{
  x += dx; y += dy;
  return true;
}
```

7.1.1 Inline

The inline directive can be included before a function declaration to specify that the function must be compiled as code in the same point where it is called. This is equivalent to declare a macro, and its advantage is only appreciated in very short functions, in which the resulting code from compiling the program may be faster if the overhead of calling a function (stacking of arguments) is avoided.

```
inline type name (arguments) statement
```

The possibility of mutually recursive inline functions, inline functions that recurse or not depending on input, etc., makes it impossible to guarantee that every call of an inline function is actually inlined.

Example

```
inline int factorial (int n) { return (n < 2) ? 1 : n * factorial(n-1); }
```

A clever compiler can generate the constant 720 for a call factorial(6). The degree of cleverness of a compiler cannot be legislated, so one compiler might generate 720, another 6*factorial(5), and yet another an un-inlined call factorial(6).

7.2 Arguments

Arguments can be passed in two ways: by value when called function receives a copy of the argument, or by reference when called function is given access to the original variable in the calling function.

Passing parameters by value (the default mechanism in C++ and the only parameter passing mechanism in C and Java) causes the creation of a copy of the passed argument. For example

```
void f(int i, string szText)
{
  cout << i << " " << szText <<  endl;
}

...

string szText = "Lasse";

for (int i = 100; i > 0; i--)
{
  f(i, szText);
}
```

In this example, the one hundred function calls create one hundred copies of the variable bev: one per call. If the prototype of the function verse is changed to use a reference parameter no copies are made.

```
void f(int i, string &szText)
```

The pass-by-reference (indicated by the & in the parameter) means that no copy is made in passing an argument.

7.2.1 Const

Functions that modify call-by-reference arguments can make programs hard to read and should most often be avoided. It can, however, be noticeable more efficient to pass a large object by reference than to pass it by value. In that case, the argument might be declared const to indicate that the reference is used for efficiency reasons only and not to enable the called function to change the value of the object.

```
void f(int i, const string &szText)
```

The const modifier means that the parameter cannot be modified within the body of function. The reference is for efficiency and the const is for safety.

7.2.2 Arrays

If an array is used as a function argument, a pointer to its initial element is passed. The size of an array is not available to the called function.

Examples

```
void f1(int* vector_ptr, int vector_size);

struct Vector {
  int* ptr;
  int size;
};

void f2(const Vector& v);
```

7.2.3 argc and argv

Sometimes it is very useful to pass information into a program when you run it. The general method is to pass information into the main() function through the use of command line arguments. The argc parameter holds the number of arguments on the command line and is an integer. It is always at least 1 because the name of the program qualifies as the first argument. The argv parameter is a pointer to an array of character pointer. Each element in this array points to a command-line argument.

Example

```
int main(int argc, char* argv[])
{
  if (argc=1) {
    cout << argv[0] << endl;
  }

  return 0;
}
```

Did You Know?

- The C and C++ standards do not specify the order of evaluation for function arguments. This can lead to subtle portability problems.

7.3 Return Values

The return statement has two important uses. First, it causes an immediate exit from the function that it is in. That is, it causes program execution to return back to the calling code. Second, it may be used to return a value.

A value must be returned from a function that is not declared void. A return value is specified by a return statement. A return statement is considered to initialize an unnamed variable of the returned type. The type of a return expression is checked against the type of te returned type, and all standard and user-defined type conversions are performed. Each time a function is called, a new copy of its arguments and local variables is created. The store is reused after the function returns, so a pointer to a local variable should never be returned.

Examples

```
// return pointer of first occurrence of c in s
char* match(char c, char* s)
{
  while (c != *s && *s) s++;
  return s;
}

int* fp()
{
  int local = 123;
  ...
  return &local; // error
}
```

7.4 Overloading

Function overloading is simply the process of using the same name for two or more functions. Each redefinition of the function must use either different types of parameters or a different number of parameters.

Examples

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

double f(double i);
int f(int i);
int f(int i, int j);

int main()
{
  cout << f(2.0) << endl;
  cout << f(2) << endl;
  cout << f(2, 3) << endl;
  return 0;
}

double f(double i)
{
  return i;
}

int f(int i)
{
  return i;
}

int f(int i, int j)
{
  return i*j;
}
```

7.4.1 Constructors

Constructor functions are no different from other type of functions. The most common reason to overload a constructor is to allow an object to be created by using the most appropriate and natural means for each particular circumstance.

7.4.2 Finding the Address

When the address of an overloaded function is assigned to a function pointer, the declaration of the pointer determines which function's address is assigned. The declaration of the function pointer must exactly match one and only one of the overloaded function's declarations.

Example

```
int f(int i);
int f(int i, int j);

void main()
{
  int (*fp)(int i); // declaration of the pointer

  fp = f;           // points to int f(int i);
  ...
}
```

7.4.3 Operators

Operator overloading is closely related to function overloading. Operators can be overloaded by creating operator functions. In C++ it is possible to overload most operators so that they perform special operation relative to created classes. When operator is overloaded, none of its original meanings are lost.

Example

```
class Counter
{
public:
  Counter(): this.i(0);
  Counter(short i): this.i(i);
  ~Counter() {}
  short getValue() const { return i; }
  void setValue(short i) { this.i = i; }
  Counter operator+(const Counter &);
private:
  short i;
}

Counter Counter::operator+(const Counter & op)
{
  return Counter(this.i + op.getValue());
}

void main()
{
  Counter co1(5), co2(2), co3;

  co3 = co1 + co2;
  cout << co3.getValue() << endl;
}
```

8 Classes

The aim of the C++ class concept is to provide the programmer with a tool for creating new types that can be used as conveniently as the built-in types. A type is a concrete representation of a concept. For example, the C++ built-in type float with its operations provides a concrete approximation of the mathematical concept of real number.

The fundamental idea in defining a new type is to separate the incidental details of the implementation from the properties essential to the correct use of it. Such a separation is best expressed by channeling all uses of the data structure and internal housekeeping routines through a specific interface.

8.1 Definition

A class declaration defines a new type that links code and data. The general form for a class declaration is

```
class name {
  private data and functions
access specifier:
  data and functions
...
access specifier:
  data and functions
} object-list;
```

Where the name is the name by which it will be possible to create an object from class, access specifier is one of three keywords: public, private, or protected, and the object-list is optional list of objects.

By default, data and functions declared within a class are private and may be accessed only by other members of the class. By using public access specifier, everyone can access data and functions. Specifying that a data member or member function is protected means that it can only be accessed from within the class or a subclass.

Example

```
class A {
public:
  int i; // public to all users of class A
protected:
  int j; // can only be used by methods in class A or its derived classes
private:
  int k; // can only be used by methods in class A
}
```

8.1.1 Constructors

A class constructor (if there is one) is called after creating a new instance of the class. Constructors are often used to set up initial values for data in the new object and to allocate space for sub-objects (e.g. for an array contained in the new object). A class can have multiple constructors, taking different type of arguments.

Example

```
class myclass {
  int a, b;
public:
  myclass() { a = 0; b = 0; }
  myclass(int i, int j) { a = i; b = j; }
  void show() { cout << a << " " << b; }
};

int main()
{
  myclass ob1, ob2(1,2);

  ob1.show();
  ob2.show();

  return 0;
}
```

8.1.2 Destructor

When delete is called on an object, the destructor function of the class (if one is defined) before destroying the object. The destructor can deallocate storage within the object (e.g. an array contained in one of its fields) or perform other clean-up operations. A class cannot have more than one destructor.

Example

```
class myclass {
  int who;
public:
  myclass(int i) { cout << "Initializing " << i << endl;
who = i; }
  ~myclass() { cout << "Destructing " << who << endl; }
} glob_ob(1);

int main()
{
  myclass local_ob(2);

  return 0;
}
```

8.1.3 Structures

C++ has elevated the role of the standard C structure. The only difference between a class and a struct is that by default all members are public in a structure and private in a class.

This seeming redundancy is justified for several reasons. First, in C, structures already provide a means of grouping data. Therefore it was a small step to allow them to include member functions. Second, because structures and classes are related, it may be easier to transport existing C programs to C++. Finally, providing two different keywords allows the definition of a class to be free to evolve.

For the sake of clarity, struct should be used when C-like structure is wanted and class when a class is wanted.

Examples

```
struct mystr {
  void setStr(char *s);
  void showStr();
private:
```

```
  char str[255];
};

class mystr {
  char str[255];
public:
  void setStr(char *s);
  void showStr();
};
```

8.1.4 Unions

Like a structure, an union declaration in C++ defines a special type of class. Unions may contain both member functions and variables. They may also include constructor and destructor functions. An union retains all of its C-like features. the most important being that all data elements share the same location in memory.

There are several restrictions when unions are used: An union cannot inherit any other classes of any type. An union cannot be a base class. An union cannot have virtual member functions. No static variables can be members of an union. An union cannot have as a member any object that overloads the = operator. No object can be a member of an union if the object has a constructor or destructor function.

Example

```
union mytypes_t {
  char c;
  int i;
  float f;
} mytypes;
```

8.1.5 Friend

Friend declarations give functions from outside a class access to class's private data and private functions, without making them public.

Example

```
class myclass {
  int a, b;
public:
  friend int sum(myclass x);
  void set_ab(int i, int j);
}

void myclass::set_ab(int i, int j);
{
  a = i;
  b = j;
}

int sum(myclass x) // not a member function of any class
{
  return x.a + x.b;
}
```

8.1.6 Static Members

Both function and data members of a class can be made static. There are restrictions placed on static

member functions. First, they may only access other static members of the class. Second, static member functions do not have a this pointer. When a member variable declaration precedes with static, only one copy of that variable will exist and all objects of the class will share that variable.

Example

```
class static_type {
  static int i;
public:
  static void init(int x) { i = x; }
  void show() { cout << i; }
};

int main()
{
  static_type::init(10); // this is perfectly valid!

  static_type x;
  x.show();

  return 0;
}
```

8.2 Objects

What is an object? In design, it is an entity in the model of the system. In source code, it is a typed variable. In compiled object code, it is an allocation of memory. In memory, it is a named portion of memory.

An object is created by creating an instance of its type - this is called instantiation. Object is defined by creating a variable of its type, or by dynamically allocating it.

8.2.1 Passing to Functions

Objects are passed to functions through the use of the standard call-by-value mechanism which means that a copy of an object is made when it is passed to a function.

Example

```
class myclass {
  int who;
public:
  void set_who(int i) { who = i; }
  myclass(int i) { cout << "Initializing " << i << endl;
who = i; }
  ~myclass() { cout << "Destructing " << who << endl; }
};

void f(myclass ob);

int main()
{
  myclass local_ob(2);

  f(local_ob);

  return 0;
}

void f(myclass ob) // copy is made but the constructor is not called
{
  ob.set_who(4);
```

```
} // destructor of the copy is called
```

8.2.2 Returning

A function may return an object to the caller.

Example

```
class myclass {
  int who;
public:
  void set_who(int i) { who = i; }
  void get_who(int i) { return who; }
};

myclass f();

int main()
{
  myclass local_ob;

  local_ob = f();

  cout << local_ob.get_who() << endl;

  return 0;
}

myclass f()
{
  myclass ob;

  ob.set_who(1);

  return ob;
}
```

8.2.3 Assignment

Assuming that both objects are of the same type, one object can be assigned to another. By default, all data from one object is assigned to the other by use of a bit-by-bit copy. However, it is possible to overload the assigment operator and define some other assigment procedure.

Example

```
class myclass {
  int who;
public:
  void set_who(int i) { who = i; }
  void get_who(int i) { return who; }
};


int main()
{
  myclass local_ob;
  myclass copy_ob;

  local_ob.set_who(1);
  copy_ob = local_ob; // assign data from local_ob to copy_ob

  cout << copy_ob.get_who() << endl;
```

```
  return 0;
}
```

8.3 Inheritance

Inheritance allows the creation of hierarchical classifications. Using inheritance, it is possible to create a general class that defines traits common to a set of related items. This class may then be inherited by other more specific classes, each adding only those things that are unique to the inheriting class. The general form for a class inheritance is

```
class derived-class-name : access base-class-name {
  // body of class
};
```

The members of the base class become members of the derived class. The access status of the base class members inside the derived class is determined by access.

Example

```
class A {
public:
  int i; // public to all users of class A
protected:
  int j; // can only be used by methods in class A or its derived classes
private:
  int k; // can only be used by methods in class A
}

class B : public A {
...        // i is again public, j is again protected
}

class C : protected A {
...        // i is now protected, j is again protected
}

class D : private A {
...        // i and j are private, so users of D cannot access
}          // them, only methods of D itself
```

Access declaration *base-class::member;* can be used to restore one or more inherited members to their original access specification.

Example

```
class base {
public:
  int i;
}

class derived : private base {
public:
  base::i; // make i public again
  ...
}
```

8.3.1 Multiple Base Classes

It is possible for a derived class to inherit two or more base classes.

Example

```
class base1 {
protected:
  int i;
public:
  void show_i() { cout << i << endl; }
};

class base2 {
protected:
  int j;
public:
  void show_j() { cout << j << endl; }
};

class derived : public base1, public base2 {
public:
  void set(int i, int j) { this.i = i; this.j =j; }
};

int main()
{
  derived ob;

  ob.set(1, 2);
  ob.show_i;
  ob.show_j;

  return 0;
}
```

When two or more objects are derived from a common base class, multiple copies of the base class can be prevented from being present in an object derived from those objects by declaring the base class as virtual when it is inherited.

Example

```
class base {
protected:
  int i;
};

class derived1 : virtual public base {
protected:
  int j;
};

class derived2 : virtual public base {
public:
  int k;
};

class derived3 : public derived1, public derived2 {
public:
  int sum;
}

int main()
{
  derived ob;

  ob.i = 1;
  ob.j = 2;
  ob.k = 3;

  ob.sum = ob.i + ob.j + ob.k;
```

```
  cout << ob.sum << endl;

  return 0;
}
```

8.3.2 Constructors and Destructors

When an object of a derived class is created, if the base class contains a constructor, it will be called first, followed by the derived class constructor. When a derived object is destroyed, its destructor is called first, followed by the base class desctructor, if it exists. In other words, constructor functions are executed in their order of derivation. Destruction functions are executed in reverse order of derivation.

Passing arguments to a constructor function in a base class can be done by using expanded form of the derived class's constructor declaration that passes along arguments to one or more base class constructors. The general form of this expanded derived class construction is

```
derived-constructor(arguments) : base1(arguments),
                                 base2(arguments),
                                 ...,
                                 baseN(arguments)
{
  // body of derived constructor
}
```

Where base1 through baseN are the names of the base classes inherited by the derived class. A colon separates the derived class's constructor function declaration from the base classes and the base classes are separated from each other by commas, in the case of multiple base classes.

Example

```
class base1 {
protected:
  int i;
public:
  base1(int i) { this.i = i; }
};

class base2 {
protected:
  int j;
public:
  base1(int j) { this.j = j; }
};

class derived : public base1, public base2 {
private:
  int k;
public:
  derived(int i, int j, int k): base1(i), base2(j) { this.k = k }
  void show() { cout << i << j << k << endl; }
};

int main()
{
  derived ob(1, 2, 3);

  ob.show;

  return 0;
}
```

8.4 Polymorphism

Polymorphism allows one interface to be used with a general class of actions. The specific action selected is determined by the type of data involved.

Polymorphism is supported by C++ both at compile time and at run time. Compile-time polymorphism is accomplished by using overloaded functions and operators. Run-time polymorphism is accomplished by using inheritance and virtual functions.

8.4.1 Virtual Functions

A virtual function is a function that is declared as virtual in a base class and redefined by a derived class.

Example

```
class base {
public:
  virtual void f() {
    cout << "This is base's function" << endl;
  }
}

class derived1 : public base {
public:
  void f() {
    cout << "This is derived1's function" << endl;
  }
}

class derived2 : public derived1 {
public:
  // no matter how many times a virtual function is inherited,
  // it remains virtual
  void f() {
    cout << "This is derived2's function" << endl;
  }
}

class derived3 : public base {
public:
// f() not overridden by derived3, base's is used
}

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;
  derived3 d3;

  p = &b;   // point to base
  p -> f(); // access base's function

  p = &d1;  // point to derived1
  p -> f(); // access derived1's function

  p = &d2;  // point to derived2
  p -> f(); // access derived2's function

  p = &d3;  // point to derived3
  p -> f(); // access base's function
```

```
  return 0;
}
```

8.4.2 Overriding

The term overriding is used to describe virtual function redefinition by a derived class. Overriding differs from overloading a normal function, in which return types and the number and type of parameters may differ. When a virtual function is redefined, all aspects of its prototype must be the same. Virtual functions must be members of a classes they are part of - they cannot be friends. Constructor functions cannot be virtual, but destructor functions can.

8.4.3 Pure Virtual Function

A pure virtual function is a virtual function that has no definition within the base class. The general form for a pure virtual function is

```
virtual type name(arguments) = 0;
```

When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile-time error will result.

Example

```
class base {
protected:
  int i;
public:
  void set_i(int i) { this.i = i; }
  virtual void show() = 0;
}

class derived : public base {
public:
  void show() { cout << i << endl; }
}

int main()
{
  derived d;

  d.set_i(1);
  d.show();

  return 0;
}
```

8.4.4 Abstract Classes

A class that contains at least one pure virtual function is said to be abstract. Because an abstract class contains one or more functions for which there is no definition, no objects may be created by using abstract class. Althought it is impossible to create objects of an abstract class, it is possible to create pointers to an abstract class. This allows abstract classes to support run-time polymorphism, which relies upon base class pointers to select the proper virtual function.

8.4.5 Binding

The term *early binding* refers to events that occur at compile time. In essence, early binding means

that all information needed to call a function is known at compile time. The main advantage to early binding is efficiency - all information necessary is determined at compile time.

The opposite of early binding is *late binding*. Late binding refers to function calls that are not resolved until run-time. Virtual functions are used to achieve late binding. The main advantage to late binding is flexibility but because a function call is not resolved until run-time, it can make slower execution times.

9 Templates

In simplest terms, a template is a definition of either class or a function that has one or more C++ types (a class or built-in type) as parameters.

Re-inventing source code is not an intelligent approach in an object oriented environment which encourages re-usability. Templates are very useful when implementing generic constructs like vectors, stacks, lists, queues which can be used with any arbitrary type. C++ templates provide a way to re-use source code as opposed to inheritance and composition which provide a way to re-use object code.

C++ provides two kinds of templates: function templates and class templates. Function templates can be used to write generic functions that can be used with arbitrary types. For example, searching and sorting routines which can be used with any arbitrary type. The Standard Template Library (STL) generic algorithms have been implemented as function templates, and the containers have been implemented as class templates.

9.1 Definition

Templates allow to create generic functions and classes that admit any data type as parameters and return value without having to overload the functions with all the possible data types. The general form for a template is

```
template <argument-list> declaration
```

Where the keyword, template, marks the location of a template definition, argument list can have "normal" function and type arguments, and the declaration is the parameterized version of a class or function.

Example

```
template <class T, int size>
class Array {
protected:
  T *data;
public:
  Array() { data = new T[size]; }
  ~Array() { delete[] data; }
  T &operator[](int index) {
    if (index < 0 || index >= size)
      throw("Bad index"); <FONT class="comment">// generate a runtime error
    else
      return data[index];
  }
};

int main()
{
  Array<int, 20> a;

  a[0] = 0;
  a[1] = 1;
  a[25] = 25; <FONT class="comment">// generates exception

  return 0;
```

```
}
```

From the point of view of the compiler, templates are not normal function or classes. They are compiled on demand. Meaning that the code of a template function is not compiled until an instantiation is required. At that moment, when an instantiation is required, the compiler generates from the template a function specifically for that type.

9.1.1 Versus Macros

Because the creation of new classes and functions takes place at compile time, it is convient to think of the template facility as a sort of macro preprocessor with superpowers. Bjarne Stroustrup says it is reasonable to think of a template as "a clever kind of macro that obeys the scope, naming, and type rules of C++."

It is difficult to write macros that work properly under all circumstances. Macros don't check argument types, generate unpleasant side effects, and can be just plain dangerous to use. Even a simple macro can cause all sorts of problems.

Example

```
#define max(a, b) a > b ? a : b
...
*p++ = max(*r++, *s++);
```

Each time macro argument (a or b) is evaluated, it will cause pointer (r or s) to be incremented. Following example shows safer approach.

Example

```
template <class GenericType>
GenericType &max(const GenericType &a, const GenericType &b)
{
  if (a > b)
    return a;
  else
    return b;
}
...
*p++ = max(*r++, *s++);
```

The macro-like functionality of templates, forces to a restriction for multi-file projects: the implementation (definition) of a template class or function must be in the same file as the declaration. It is not possible to separate the interface in a separate header file and both interface and implementation must be included in any file that uses the templates.

9.2 Types

The two major types of templates are function templates and class templates. Both act as factories for their particular type of objects. That is, class templates generate classes and function templates generate functions. Function templates provide a way to write a single function definition where the data type is a parameter.

9.2.1 Function

A function template specifies an unbounded set of (overloaded) functions. A function generated from a function template is called a template function, as is a function defined with a type that matches a

function template.

Example

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

template <class GenericType>
void ConvertFToC(GenericType f, GenericType &c);

int main()
{
  double df, dc;
  float ff, fc;
  int i_f,i_c;

  df = 75.0;
  ff = 75.0;
  i_f = 75;

  ConvertFToC(df,dc);
  cout << df << " == " << dc << endl;

  ConvertFToC(ff,fc);
  cout << ff << " == " << fc << endl;

  ConvertFToC(i_f,i_c);
  cout << i_f << " == " << i_c << endl;
}

template <class GenericType>
void ConvertFToC(GenericType f, GenericType &c)
{
  c = (f - 32.0) * 5. / 9.;
}
```

9.2.2 Class

A class template specifies how individual classes can be constructed much as a class declaration specifies how individual objects can be constructed.

Example

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

template <class GenericType, int size>
class Stack {
public:
  Stack() : index(-1) {}
  ~Stack() {}
  void push(GenericType val);
  GenericType pop();
private:
  GenericType data[size];
  int index;
};

// Method definitions
template <class GenericType, int size>
void Stack<GenericType, size>::push(GenericType val)
{
  data[++index] = val;
}
```

```
template <class GenericType, int size>
T Stack<GenericType, size>::pop()
{
  return data[index--];
}

int main()
{
  int val;

  Stack<int, 100> stack1;
  Stack<float, 10> stack2;

  stack1.push(1);
  stack1.push(2);
  stack2.push(1.1);
  stack2.push(2.2);

  val = stack1.pop();
  cout << "popped " << val << endl;
  val = stack1.pop();
  cout << "popped " << val << endl;

  val = stack2.pop();
  cout << "popped " << val << endl;
  val = stack2.pop();
  cout << "popped " << val << endl;

  return 0;
}
```

9.3 STL

The Standard Template Library (STL) is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template.

9.3.1 Containers

A Container is an object that stores other objects (its elements), and that has methods for accessing its elements. In particular, every type that is a model of Container has an associated iterator type that can be used to iterate through the Container's elements.

Like many class libraries, the STL includes container classes. The STL includes the classes vector, list, deque, set, multiset, map, multimap, hash_set, hash_multiset, hash_map, and hash_multimap. Each of these classes is a template, and can be instantiated to contain any type of object.

Examples

```
vector<int> V;
V.insert(V.begin(), 3);
assert(V.size() == 1 && V.capacity() >= 1 && V[0] == 3);

deque<int> Q;
Q.push_back(3);
Q.push_front(1);
Q.insert(Q.begin() + 1, 2);
Q[2] = 0;
copy(Q.begin(), Q.end(), ostream_iterator<int>(cout, " "));
// The values that are printed are 1 2 0
```

```
list<int> L;
L.push_back(0);
L.push_front(1);
L.insert(++L.begin(), 2);
copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));
// The values that are printed are 1 2 0
```

9.3.2 Algorithms

The STL includes a large collection of algorithms that manipulate the data stored in containers.

Examples

```
list<int> L;
L.push_back(3);
L.push_back(1);
L.push_back(7);
list<list<int>::iterator result = find(L.begin(), L.end(), 7);
assert(result == L.end() || *result == 7);

int A[] = {1, 4, 2, 8, 5, 7};
const int N = sizeof(A) / sizeof(int);
sort(A, A + N);
copy(A, A + N, ostream_iterator<int>(cout, " "));
// The output is " 1 2 4 5 7 8"
```

9.3.3 Iterators

Iterators are a generalization of pointers: they are objects that point to other objects. As the name suggests, iterators are often used to iterate over a range of objects: if an iterator points to one element in a range, then it is possible to increment it so that it points to the next element.

Iterators are central to generic programming because they are an interface between containers and algorithms: algorithms typically take iterators as arguments, so a container need only provide a way to access its elements using iterators. This makes it possible to write a generic algorithm that operates on many different kinds of containers, even containers as different as a vector and a doubly linked list.

Example

```
// copy the elements of a vector to the standard output, one per line.
vector<int> V;
// ...
copy(V.begin(), V.end(), ostream_iterator<int>(cout, "\n"));
```

10 Exception Handling

The C++ language provides built-in support for handling anomalous situations, known as "exceptions," which may occur during the execution of a program. With C++ exception handling, a program can communicate unexpected events to a higher execution context that is better able to recover from such abnormal events. These exceptions are handled by code that is outside the normal flow of control.

The global concept of exception handling is simple. The idea is to raise some error flag every time something goes wrong. Next, there is a system that is always on the lookout for this error flag. Third, the previous system calls the error handling code if the error flag has been spotted.

10.1 Definition

Exception handling provides a way for a function that encounters an unusual situation to throw an exception and pass control to a direct or indirect caller of that function. The caller may or may not be able to handle the exception. Code that intercepts an exception is called a handler. Regardless of whether or not the caller can handle an exception, it may rethrow the exception so it can be intercepted by another handler.

The general form of use for an exception is

```
try {
  // code to be tried
  throw exception;
}
catch (type exception)
{
  // code to be executed in case of exception
}
```

The code within the try block is executed normally. In case that an exception takes place, this code must use throw keyword and a parameter to throw an exception. The type of the parameter details the exception and can be of any valid type. If an exception has taken place, that is to say, if it has been executed a throw instruction within the try block, the catch block is executed receiving as parameter the exception passed by throw.

Example

```
int main () {
  try
  {
    char* mystring;
    mystring = new char [10];
    if (mystring == NULL) throw "Allocation failure";
    for (int n = 0; n <= 100; n++)
    {
      if (n>9) throw n;
      mystring[n] = 'z';
    }
  }
  catch (int i)
  {
    cout << "Exception: ";
    cout << "index " << i << " is out of range" <<
```

```
endl;
  }
  catch (char* str)
  {
    cout << "Exception: " << str << endl;
  }
  return 0;
}
```

If an exception is not caught by any catch statement because there is no catch statement with a matching type, the special function terminate will be called. This function is generally defined so that it terminates the current process immediately showing an "Abnormal termination" error message.

10.1.1 Standard Exceptions

Some functions of the standard C++ language library send exceptions that can be captured if we include them within a try block. These exceptions are sent with a class derived from std::exception as type. This class (std::exception) is defined in the C++ standard header file <exception> and serves as pattern for the standard hierarchy of exceptions:

- bad_alloc (thrown by new)
- bad_cast (thrown by dynamic_cast when fails with a referenced type)
- bad_exception (thrown when an exception doesn't match any catch)
- bad_typeid (thrown by typeid)
- logic_error
  - domain_error
  - invalid_argument
  - length_error
  - out_of_range
- runtime_error
  - overflow_error
  - range_error
  - underflow_error
- ios_base::failure (thrown by ios::clear)

Because this is a class hierarchy, if you include a catch block to capture any of the exceptions of this hierarchy using the argument by reference (i.e. adding an ampersand & after the type) you will also capture all the derived ones (rules of inheritance in C++).

Example

```
#include <iostream.h>
#include <exception>
#include <typeinfo>

class A {virtual f() {}; };

int main () {
  try {
    A* a = NULL;
    typeid (*a); // throws bad_typeid
  }
  catch (std::exception& e)
  {
    cout << "Exception: " >> e.what();
  }
  return 0;
}
```

10.2 Handling System

In the following pseudo code, Exception is a defined class with a constructor with no parameters (as identified by the throw-call).

```
try {
  ...
  throw Exception()
  ...
} catch( Exception e )
{
  ...
}
```

It would be useful to have some info on what kind of error occurred. This could be done in two ways: By defining different exception-classes and throw them according to which error occurred or by giving the class as a parameter containing an error message and allow the class to display the message.

The class should store info about the error that occurred and the class should able to display an error message. The report function will show the error some way.

```
class CException {
public:
  char* message;
  CException( char* m ) { message = m };
  Report();
}
```

With the given class the power of exception handling can be really shown off.

```
try {
  Initialize();
  Run();
  Shutdown();
} catch(CException e)
{
  e.Report();
}
```

The functions Initialize() and Run() should be considered heavily errorous and with several sub-function-calls that could throw exceptions as well. Now all occurring errors throughout the whole program will be handled by this single catch-statement and will be shown on the display with the single Report()-statement.

11 Input and Output

I/O, input and output, form an important part of any program. To do anything useful a program needs to be able to accept input data and report back results.

C++ defines an I/O system providing complete support for object-oriented programming that can operate on user defined objects. C++'s I/O system is fully integrated - the different aspects of C++'s I/O system such as console I/O and file I/O, are actually just different perspectives on the same mechanism.

11.1 Console

In C++, input and output are provided by the iostream library. To use this library, **#include <iostream>** must be added to the top of a program. This tells the preprocessor to add code from the file iostream.h into a source file. Including this file defines and initializes the following objects for use in a program.

Table 11.1: Input and output objects.

Object	Description
cin	provides for input from the terminal (keyboard)
cout	provides for output to the screen
cerr	provides unbuffered output to the standard error device, which defaults to the screen. Unbuffered means that any messages or data will be written immediately. With buffered input, data is saved to a buffer by the operating system, transparently to your program. When the buffer is full, everything in it is written out. This is more efficient because each write requires a certain amount of overhead from the operating system. Writing out one large buffer has less overhead than writing out multiple smaller messages. The downside is that if a program crashes before the buffer is written, nothing in the buffer is output. Output via cerr is unbuffered to ensure that error messages will be written out.
clog	provides buffered output to the standard error device, which defaults to the screen

Example

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

int main()
{
  string name;
  int ID;

  cout << "Enter your name ";
  cin >> name;

  cout << "Enter your ID number ";
  cin >> ID;

  cout << "Hello " << name << " or should I say "
<< ID << endl;

  return 0;
}
```

11.1.1 Format Flags

Associated with each stream is a set of format flags that control some of the ways information is formatted by a stream.

Table 11.2: Format flags.

Flag	Description
skipws	leading white-space characters (spaces, tabs, and newlines) are discarded when input is performed on a stream.
left	output is left justified.
right	output is right justified.
internal	a numeric value is padded to fill a field with spaces inserted between any sign or base character.
dec	numeric values are output in decimal (default).
oct	output is displayed in octal.
hex	output is displayed in hexadecimal.
showbase	the base of numeric values is shown.
showpoint	a decimal point and trailing zeroes are displayed for all floating-point output.
uppercase	characters are displayed in uppercase.
showpos	a leading plus sign is displayed before positive values.
scientific	floating-point numeric values are displayed in scientific notation.
fixed	floating-point values are displayed in normal notation.
unitbuf	I/O system performance is imporved because output is partially buffered.
stdio	stream is flushed after each output.

Example

```
#include <iostream>

void showFlags();

int main()
{
  showFlags();

  cout.setf(ios::right | ios::showpoint | ios::fixed);

  showFlags();

  return 0;
}

void showFlags()
{
  long f, i;
  int j;

  char flags[15][12] = {
    "skipws", "left", "right", "internal", "dec", "oct", "hex",
    "showbase", "showpoint", "uppercase", "showpos", "scientific",
    "fixed", "unitbuf", "stdio"
  };

  f = cout.flags();

  for (i = 1, j = 0; i <= 0x4000; i = i<<1, j++)
    if (i & f) cout << flags[j] << " is on" << endl;
    else cout << flags[j] << " is off" << endl;
```

```
}
```

11.1.2 Format Methods

In addition to the formatting flags, there are three member functions defined by ios that set these format parameters: the field with, the precision, and the fill character.

Table 11.3: Format methods.

Method	Description
int width(int w);	w becomes the field width, and the previous field width is returned.
int precision(int p);	the precision is set to p, and the old value is returned.
char fill(char ch);	ch becomes the new fill character, and the old one is returned.

Example

```
#include <iostream>

int main()
{
  cout.precision(d2);
  cout.width(10);

  cout << 1.23456 << endl; // displays 1.23

  cout.fill('#');

  cout.width(10);
  cout << 1.23456 << endl; // displays ######1.23

  cout.width(10);
  cout.setf(ios::left);
  cout << 1.23456 << endl; // displays 1.23######
}
```

11.1.3 Overloading Inserts

All inserter functions have a general form:

```
ostream &operator<<(ostream &stream, class_type obj)
{
  // body of inserter
  return stream;
}
```

The function returns a reference to a stream of type ostream. The first parameter to the function is a reference to the output stream. The second parameter is the object being inserted. Finally, the inserter is returning stream which allows the inserter to be used in a chain of insertions.

Example

```
#include <iostream>

class box {
private:
  int x, y;
public:
  box(int x, int y) { this.x = x; this.y = y; }
  friend ostream &operator<<(ostream &stream, box o);
}

ostream &operator<<(ostream &stream, box o)
{
```

```
  register int i, j;

  for (i = 0; i < o.x; i++)
    stream << "*";

  stream << endl;

  for (j = 1; j < o.y - 1; j++) {
    for (i = 0; i < o.x; i++)
      if (i == 0 || i == o.x - 1) stream << "*";
      else stream << " ";
    stream << endl;
  }

  for (i = 0; i < o.x; i++)
    stream << "*";

  stream << endl;
  return stream;
}

int main()
{
  box a(5, 5), b(8, 8), c(12, 17);

  cout << a << b << c;

  return 0;
}
```

11.2 File

The techniques for file I/O in C++ are virtually identical to those introduced in console for writing and reading to the standard output devices, the screen and keyboard. To perform file input and output the include file fstream must be used.

Fstream contains class definitions for classes used in file I/O. Within a program needing file I/O, for each output file required, an object of class ofstream is instantiated. For each input file required, an object of class ifstream is instantiated. The ofstream object is used exactly as the cout object for standard output is used. The ifstream object is used exactly as the cin object for standard input is used.

Classes ofstream, ifstream and fstream are derived from ostream, istream and iostream respectively. That's why fstream objects can use the members of these parent classes to access data.

11.2.1 Opening and Closing a File

Before a file can opened, a stream must be obtained. There are three type of streams: input, output, and input/output.

```
ifstream in;  // input
ofstream out; // output
fstream io;   // input and output
```

Once a stream is created, one way to associate it with a file is by using the function open() which is a member of each of the three stream classes. Prototype of function open() is

```
void open(char* filename, int mode, int access);
```

Where filename is the name of the file, which may include a path specifier. The value of mode determines how the file is opened (see table 11.4). The value of access determines how the file can be

accessed (see table 11.5).

Table 11.4: Mode values.

Mode	Description
ios::app	causes all output to a file to be appended to the end. This value can be used only with files capable of output.
ios::ate	causes a seek to end-of-file to occur when the file is opened.
ios::in	specifies that the file is capable of input.
ios::out	specifies that the file is capable of output.
ios::noreplace	causes the open() function to fail if the file does already exist.
ios::nocreate	causes the open() function to fail if the file does not already exist.
ios::trunc	causes the contents of a preexisting file by the same to be destroyed and truncates the file to zero length.

Table 11.5: Access values.

Value	Description
0	Normal file - open access
1	Read-only file
2	Hidden file
4	System file
8	Archive bit set

Example

```
fstream mystream1;
mystream1.open("test", ios::in | ios::out, 0);
if (!mystream1) {
  cout << "Cannot open file" << endl;
  // handle error
}
mystream1.close();

fstream mystream2("test", ios::in | ios::out, 0);
if (mystream2.fail()) {
  cout << "Cannot open file" << endl;
  // handle error
}
mystream2.close();
```

11.2.2 Reading and Writing Text Files

When reading from or writing to a text file, the << and >> operators can be used the same way when performing console I/O, except that instead of using cin and cout, a stream that is linked to a file is used.

Example

```
// Writing to a file
#include <iostream>
#include <fstream>
using namespace std;

#define FILENAME "name.txt"

int main()
{
```

```
  ofstream out(FILENAME);

  if (!out) {
    cout << "Cannot open " << FILENAME  << " file."
<< endl;
    return 1;
  }

  out << "Lasse" << endl;
  out << "Markus" << endl;

  out.close();
  return 0;
}
```

Example

```
// Reading from a file
#include <iostream>
#include <fstream>
using namespace std;

#define FILENAME "name.txt"

int main()
{
  ifstream in(FILENAME);

  if (!in) {
    cout << "Cannot open " << FILENAME  << " file."
<< endl;
    return 1;
  }

  char name[10];

  while (in >> name)
    cout << name;

  in.close();
  return 0;
}
```

11.2.3 Reading and Writing Binary Files

There are who ways to write and read binary data to or from a file. First, by using the member function **put()** and read a byte by using the member function **get()**. The second way to read and write blocks of binary data is to use **read()** and **write()** functions. Prototypes of these functions are

```
istream &get(char &ch);
ostream &put(char ch);
istream &read(unsigned char* buf, int num);
ostream &write(const unsigned char* buf, int num);
```

The **get()** function reads a single character from the associated stream and puts that value in ch. It returns a reference to the stream. The **put()** function writes ch to the stream and returns the stream. The **read()** function reads num bytes from the associated stream and puts them in the buffer pointed to by buf. The **write()** function writes num bytes to the associated stream from the buffer pointed to by buf.

Example

```
// get() and put()
```

```
#include <iostream>
#include <fstream>

const char* filename = "chars.txt";

int main () {
  ofstream out(filename);
  if (!out) {
    cout << "Cannot open output file." << endl;
    return 1;
  }

  int i;
  // write all characters to a file
  for (i = 0; i < 256; i++) out.put(i);

  out.close();

  ifstream in(filename);
  if (!in) {
    cout << "Cannot open file." << endl;
    return 1;
  }

  while (in) { // in is 0 when eof is reached
    in.get(ch);
    cout << ch;
  }

  in.close();

  return 0;
}
```

Example

```
// read() and write()
#include <iostream>
#include <fstream>
#include <string>

const char* filename = "balance.txt";

struct status {
  char name[80];
  float balance;
  unsigned long account_num;
};

int main()
{
  struct status acc;

  strcpy(acc.name, "Lasse Rautiainen");
  acc.balance = 12345.67;
  acc.account_num = 12345678;

  // write
  ofstream outbal(filename);

  if (!outbal) {
    cout << "Cannot open file." << endl;
    return 1;
  }

  outbal.write((unsigned char*) &acc, sizeof(struct status));
  outbal.close();
```

```
    // read
    ifstream inbal(filename);

    if (!inbal) {
        cout << "Cannot open file." << endl;
        return 1;
    }

    inbal.read((unsigned char*) &acc, sizeof(struct status));

    cout << acc.name << endl;

    return 0;
}
```

11.2.4 Passing Streams to Functions

File streams must be passed to functions by reference, not by value.

```
void myfunction(ifstream &fp, ...) // correct
void myfunction(ifstream fp, ...)  // incorrect
```

If streams are passed by value, the C++ compiler will not complain. However, mysterious bad things will start happening, often in parts of the code which don't appear to be related to the offending function.

Appendix: C/C++ Keywords

Table 1 lists the 32 keywords that, combined with the formal C syntax form the C programming language.

Table 1: The 32 keywords Defined by the ANSI C Standard

Keyword	Description
auto	Auto is the default storage class for local variables.
break	Allows the program to escape from for, while, do...while loops and switch structures.
case	A multi-way decision statement.
char	Defines characters.
const	Used to tell C that the variable value can not change after initialisation.
continue	Allows a new iteration of a loop without the current iteration completing.
default	An optional clause that is matched if none of the constants in the case statements can be matched.
do	Repeats a block of statements.
double	Used to define BIG floating point numbers.
else	A two-way decision statement.
enum	Allows to define a list of aliases which represent integer numbers.
extern	Defines a global variable that is visable to ALL object modules.
float	Used to define floating point numbers.
for	Used to repeat a block of code many times.
goto	Allows the program to 'jump' to a named label (never required).
if	A two-way decision statement.
int	Used to define integer numbers.
long	Data type modifier.
register	Used to define local variables that should be stored in a register instead of RAM.
return	Will return a value from a function to its caller.
short	Data type modifier.
signed	Data type modifier.
sizeof	Will return the number of bytes reserved for a variable or data type.
static	The default storage class for global variables.
struct	Used to declare a new data-type.
switch	A multi-way decision statement.
typedef	Used to define new data type names to make a program more readable to the programmer.
union	Allows several variables of different type and size to occupy the same storage location.
unsigned	Data type modifier.
void	Allows us to create functions that either do not require any parameters or do not return a value.
volatile	The volatile keyword acts as a data type qualifier.
while	Repeatedly executes a block of statements.

In addition the C++ extensions to C add the keywords shown in table 2.

Table 2: The C++ keywords

Keyword	Description
asm	Used to add assembly language into the source code, implementation specific.

Keyword	Description
catch	Used with error handling to catch exceptions.
class	To classify objects in terms of data and behaviour.
delete	Used to free memory.
friend	Used to declare functions as friends of the class. Friend functions are not part of the class, but have access to the private members.
inline	A request to the compiler to treat a function as a macro.
new	Used to allocate memory.
operator	Used to overload existing operators.
private	Implements data hiding by defining which parts of the class are not accessible from outside the class.
protected	Allows members to be accessed either by the class, or classes derived from this class.
public	Defines which parts of the class are accessible from outside the class.
template	Allows the types of class and function arguments to be parameterised.
this	An implicitly defined constant pointer for all members in a class, where the type of this is the class itself.
throw	Used with error handling to throw an exception.
try	Used with error handling, a block of code to try that may cause an exception error.
virtual	Used to implement late or dynamic binding with overloaded functions.

Did You Know?

- Original version of C defined 27 keywords. The ANSI committee added keywords **enum**, **const**, **signed**, **void**, and **volatile**.

Appendix: Precedence

Each operator has a specified precedence. In the following table, all the operators are summarized in order by priority. When several operators are grouped together, they share the same precedence. Higher precedence are done before lower precedence. Left to right among equal precedence except: unary, assignment, and conditional operators.

The meanings presented here apply when the operands are of built-in types.

Table 3: C++ Operators by Precedence

Priority	Operator	Description
1	class_name :: member	scope resolution
	namespace_name :: member	scope resolution
	:: name	global
	:: qualified-name	global
2	object . member	member selection
	pointer -> member	member selection
	pointer [expr]	subscripting
	expr (expr_list)	function call
	type (expr_list)	value construction
	lvalue ++	post increment
	lvalue --	post decrement
	typeid (type)	type identification
	typeid (expr)	run-time type identification
	dynamic_cast < type > (expr)	run-time checked conversion
	static_cast < type > (expr)	compile-time checked conversion
	reinterpret_cast < type > (expr)	unchecked conversion
	const_cast < type > (expr)	const conversion
3	sizeof expr	size of object
	sizeof (type)	size of type
	++ lvalue	pre increment
	-- lvalue	pre decrement
	~ expr	complement
	! expr	not
	- expr	unary minus
	+ expr	unary plus
	& lvalue	address of
	* expr	reference
	new type	create (allocate)
	new type (expr-list)	create (allocate and initialize)
	new (expr-list) type	create (place)
	new (expr-list) type (expr-list)	create (place and initialize)
	delete pointer	destroy (de-allocate)
	delete [] pointer	destroy array
	(type) expr	cast (type conversion)

Priority	Operator	Description
4	object .* pointer-to-member	member selection
	pointer ->* pointer-to-member	member selection
5	expr * expr	multiply
	expr / expr	divide
	expr % expr	modulo (remainder)
6	expr + expr	add (plus)
	expr - expr	subtract (minus)
7	expr << expr	shift left
	expr >> expr	shift right
8	expr < expr	less than
	expr <= expr	less than or equal
	expr > expr	greater than
	expr >= expr	greater than or equal
9	expr == expr	equal
	expr != expr	not equal
10	expr & expr	bitwise AND
11	expr ^ expr	bitwise exlusive OR
12	expr | expr	bitwise inclusive OR
13	expr && expr	logical AND
14	expr || expr	logical inclusive OR
15	lvalue = expr	simple assignment
	lvalue *= expr	multiply and assign
	lvalue /= expr	divide and assign
	lvalue %= expr	modulo and assign
	lvalue += expr	add and assign
	lvalue -= expr	subtract and assign
	lvalue <<= expr	shift left and assign
	lvalue >>= expr	shift right and assign
	lvalue &= expr	AND and assign
	lvalue |= expr	inclusive OR and assign
	lvalue ^= expr	exlusive OR and assign
16	expr ? expr : expr	Conditional expression
17	throw expr	throw exception
18	expr , expr	Comma (sequencing)

References

The C++ Programming Language, Bjarne Stroustrup, Addison Wesley, 1997.

C++ Complete Reference, Herbert Schildt, Osborne McGraw-Hill, 1991.

The C Programming Language, Brian W. Kernighan, Dennis M. Ritchie, AT&T Bell Laboratories, Second Edition.

C++ Components and Algorithms, Scott Robert Ladd, M&T Publishing, 1992.

Teach Yourself C++, Jesse Liberty, Sams Publishing, 1994.

C++ Inside, Ivor Horton, Wrox Press, 1998.

C++ Programmer's Guide to the Standard Template Library, Mark Nelson, IDG Books Worldwide, 1995.

Complete C++ language tutorial, http://www.cplusplus.com/doc/tutorial/

Standard Template Library Programmer's Guide, Hewlett-Packard Company, 1994, http://www.sgi.com/tech/stl/index.html