

C++ Programming

The Core Language

Morgan Stanley Technology Analyst Program
Class of August 2010



mallonassociates

Copyright © 1994-2010 Mallon Associates International Limited

Printed in New York, London, Mumbai & Shanghai

Developed by David Mallon

All rights reserved. No part of this work covered by copyright hereon may be reproduced in any form or by any means; graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owners.

TRADEMARKS

Sun, Sun Microsystems, the Sun Logo, Solaris and NFS are trademarks or registered trademarks of Sun Microsystems, Inc. X Window System is a trademark and product of the Massachusetts Institute of Technology. Microsoft, MS-DOS, Windows, Windows NT, the Windows and Windows NT logos, Win32 and Windows 95 are trademarks or registered trademarks of Microsoft Corporation. Unix is a registered trademark of the X/Open Company Limited. The Mallon Associates logo, Mallon Associates and the quill pen quality mark are trademarks of Mallon Associates International Limited.

All other products or services mentioned in this document are identified by the trademarks or service marks of their associated companies and are respectfully acknowledged

LIMITATIONS

Whilst ever effort has been made to ensure the accuracy, adequacy and completeness of the information and material contained in this course, the course is provided “as is”, without warranty of any kind, express or implied.

Contents

1. Introduction
2. Writing a Simple Program
3. Intrinsic data Types
4. Operators & Expressions
5. Conditional Flow Statements
6. Looping Constructs
7. Functions & Program Structure
8. Pointers & Arrays
9. Object Storage
10. class & struct
11. Constructors & Destructors
12. Object Modelling
13. Class Data & Functions
14. Operator Overloading
15. Single Inheritance
16. Polymorphism & Dynamic Binding
17. Multiple Inheritance
18. Namespaces
19. Exception Handling
20. Templates

- A. Debugging with gdb

Chapter 1: Introduction

- **What is C++?**
- **Facilities in C++**
- **Object Orientation in C++**
- **C++ Design Notes**
- **A Brief History of C++**

What is C++?

- **C++ is a general purpose programming language developed in the early 1980s by Bjarne Stroustrup.**



Supports procedural programming and is a superset of C



Supports object-oriented programming allowing users to define new data types.



Supports generic programming through class and function templates

C++ is the successor to the C programming language, and was invented by Bjarne Stroustrup at AT&T Bell Laboratories in the early 1980s.

C++ extends C with the object-oriented concepts of classes, inheritance and dynamic (run-time) type binding.

A Super-set of C

In most respects C++ is a super-set of C, although in areas where complete compatibility with C would compromise the integrity of C++ programs (for example, with the smooth integration of user defined and intrinsic types), the languages differ.

C++ Features

- **intrinsic data types**
- **conditional statements**
- **looping constructs**
- **functions**
- **modularisation**
- **low-level machine access**
- **strong typing**
- **separate compilation & linking**
- **user-defined abstract data types (OO)**
- **promotes re-use through class libraries**
- **supports generic programming**
- **provides mechanisms for structured error handling**

C++ provides all the features of a high-level programming language, including intrinsic data types, conditional and iteration statements, and the ability to modularise through files and functions.

In addition C++ maintains C's ability to access the machine at low-levels (and is therefore suitable for systems programming). Most importantly, C++ provides a wealth of facilities to support object oriented programming.

C++ Design Notes

- **Simplicity**
 - language simplicity above compiler simplicity
 - language is large, much bigger than C
- **Minimal run-time overhead**
 - heap allocation operators
 - exception handling mechanism
- **Compatibility with C**
 - ANSI C has adopted some aspects of C++
- **Total integration of user-defined types**
 - implicit and explicit casts
 - operator overloading
 - stream I/O system

The basic C++ language has no high-level data types beyond those available in C, such as `strings` or complex numbers. However, C++ was designed to enable programmers to extend the languages with their own types. The current ANSI standard has a variety of libraries of high-level types, includes `string` and `complex`.

A driving design rationale for C++ was to allow the proper integration of user defined types into the language. That is, to allow objects of user-defined types to be used where ever objects of intrinsic (in-built) types may be used. Consequently, the language supports user-defined construction, casting mechanisms and operator overloading. Even C's standard I/O library has been completely replaced since it is limited to the intrinsic types.

Run-Time Costs

Except for memory allocation operators and exception handling, there is no hidden run-time support for C++ expressions and statements.

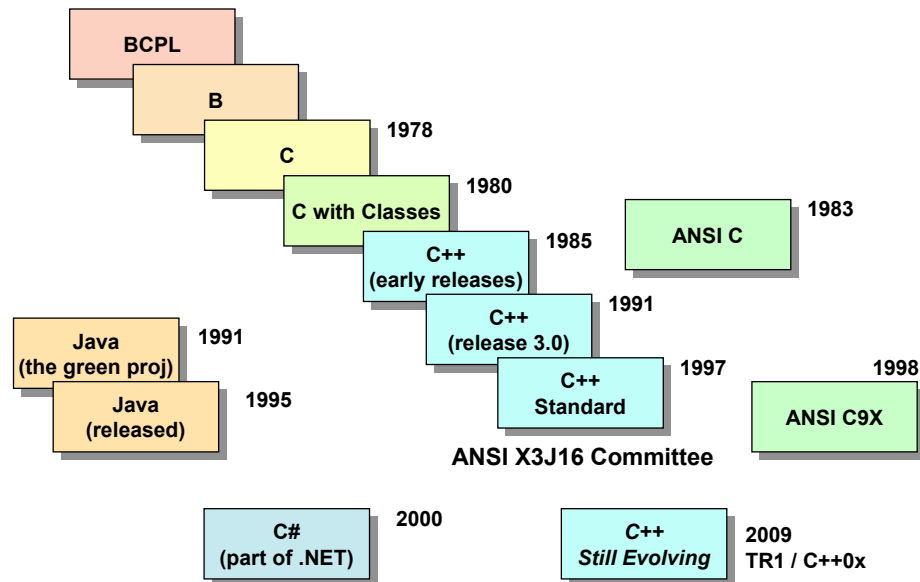
This is an important aspect of the design since it enables the OO facilities of C++ to be exploited for real programs without undue performance overhead. Most other OO languages introduce considerable run-time overheads.

Compatibility with ANSI C

Maintaining compatibility between C and C++ was important because of the existing base of code (millions of lines of libraries and functions) and knowledge (C is one of the most widely used languages). Although C++ is not entirely consistent with C, it is syntactically similar and link compatible.

Interestingly, some aspects of ANSI C are based on C++. Most notably the syntax used for function prototyping.

A Brief History



C++ was born out of C and has been designed to remain largely consistent with C. C was itself derived from a language called B, and ultimately from BCPL. All these languages have therefore contributed to C++.

In addition, the class concept was inspired by Simula67 and operator overloading by Algol68. Algol's freedom to place declarations where ever a statement may occur was also adopted. Template facilities were inspired by Ada's generics and Clu's parameterised modules; and the exception handling mechanism by Ada, Clu and ML.

The language was originally called "C with Classes" and was developed in the early 1980s. However, this language lacked many of the facilities currently available in C++, such as operator overloading, references, virtual functions, exception handling and templates.

The name C++ is derived from C's post increment operator. It was suggested that the language should be called 'P' (the next letter in BCPL), but this was rejected because otherwise C++'s strong relationship with C may be obscured.

In 1989 the X3J16 committee of ANSI was formed to standardise the language. Towards the end of 1997 the standardisation committee approved the language specification.

Not that standardisation should necessarily suggest stability. The ISO/ANSI committees have started to tinker with C once again; adding new operators and even a complex number type! The ISO 9899:199409 standard captures ANSI C as modified by Normative Addendum 1; current proposals (C9X) are still in discussion.

Syntax error

C+

Connoisseurs prefer
prefix semantics

++C

Course Notes



Chapter 2: Writing a Simple Program

- Program Structure
- main() and Compound Blocks
- Basic Types
- Expressions
- A Conditional Statement
- A Function
- C++ Development Environment

Program Structure

- A C++ program consists of a collection of

- functions - units of computation

```
print(), read(), invoice(), sort()
```

- variables - storage for data

```
x, y, total, average
```

- expressions - calculations

```
x + y / average
```

- statements - language sentences

```
total = x + y / average;
```

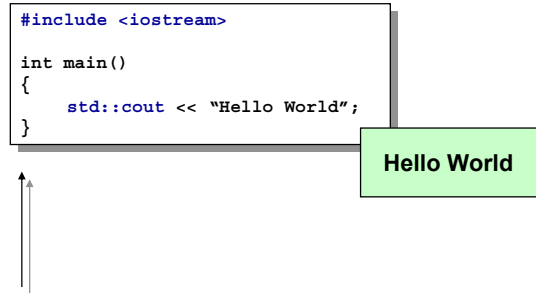
C++ has inherited the block structure of C. Programs consist of one or more functions which encapsulate a collection (block) of statements.

Statements are synonymous to conventional sentences. Each states a particular action to be performed; for example, the evaluation of an expression, declaration of a variable or call to another function. A statement is the smallest independent unit in a C++ program, and is always delimited by a semi-colon.

Statements are made up of expressions. Expressions describe the arithmetic or conditional calculations to be made, and are themselves composed of variables and values. Variables are simply references to memory, names of locations in which values have been stored.

main() and Compound Blocks

- Program execution begins at a function called main()



The braces delimit the statements in a function.

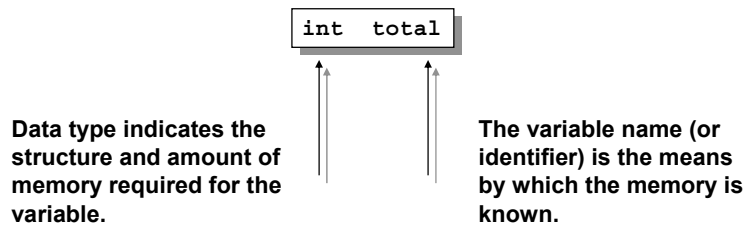
When a C++ program is executed control is passed directly to `main()`. This function (which must always appear in a C++ program) executes each of its statements in turn. The body of the function is delimited by braces.

The above program consists of a single statement to print the text "hello world". In this statement, `cout` is the output destination associated with the user's terminal. `<<` is the output operator. The statement writes to `cout` (that is, the user's terminal) the collection of characters stored within the double-quotes.

`iostream` is a standard header file and contains information about `cout` necessary for our program. `#include` is referred to as a pre-processor directive and causes the contents of the `iostream` header file to be included into the program text file. Items within `iostream` are placed in the `std` namespace and consequently scoped to this namespace when used.

Basic Types

- All variables must be declared before use



- Variables may only be used in expressions appropriate for their type

C++ is a strongly typed language. This means that all entities in a program (variables and functions) must be typed. That is, a declaration must be made to the compiler to state the kind of information which they store.

C++ comes with a number of basic (or intrinsic) types built-in. (It also offers excellent facilities for user-defined types as is fundamental in object-oriented languages.) One intrinsic type is `int`. This describes a piece of memory suitable for holding an integer value.

When declaring a variable, an identifier (or name) must be provided so that the storage can be referenced.

C++ requires that all variables are declared before they are used. This enables the compiler to verify that the use is appropriate for the given type. For example, it is not possible to store a floating point number (a decimal number) in an integer location.

Expressions

- An expression is an action involving an operator and operand(s).

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    int x;
```

```
    int y;
```

```
    int result;
```

```
    std::cout << "Please enter two numbers: ";
```

```
    std::cin >> x;
```

```
    std::cin >> y;
```

```
    result = x + y;
```

```
    std::cout << "The sum is: ";
```

```
    std::cout << result;
```

```
}
```

Please enter two numbers: 1 4

The sum is: 5



How many expressions are here?

Most statements in C++ are expressions, with the most notable exception being a declaration.

Expressions describe the action to be performed and consist of an operator and one or more operands. Expressions always evaluate to some result, and may therefore form part of other expressions. For example, the statement below

2 + 3 + 4

consists of two expressions. The first involves the addition of the operands 2 and 3; the second the addition of 5 and 4.

C++ is a rich language offering a large number of operators for arithmetic, relational, logical and bit-wise operations. In addition, it allows users to define the meaning of operators for their own objects.

In the above example there are three variable declarations and seven expressions. All are statements and therefore terminated with a semi-colon. Note, however, that there is no semi-colon at the end of the block.

This example also demonstrates how to input data into the program. `cin` is the input source associated with the user's terminal, `>>` is the input operator, and `x` and `y` are the variables into which the data are read.

Expressions

- The previous program can be written more succinctly

```
#include <iostream>

int main()
{
    int x, y, result;

    std::cout << "Please enter two numbers: ";
    std::cin >> x >> y;
    result = x + y;
    std::cout << "The sum is: " << result;
}
```

chain declarations using commas
chain expressions using operators

The previous program may be simplified. Where multiple variables of the same type must be declared, a comma separated list can be used.

Since expressions evaluate to a result, they may be used in other expressions. Hence several can be chained together.

A Conditional Statement

- Conditional Expressions

```
a > b  
total == 4
```

non-zero is true
zero is false

- if statement

```
if (conditional expression)  
    statement;  
else  
    statement;
```

← success (true)

← failure (false)

- New standard contains `bool` type

Conditional expressions determine whether something is true or false. A number of operators exist to determine the relationships between operands. C++ (like C) uses the integer value 0 to indicate falsehood or failure, and non-zero to indicate truth or success. Additionally, a `bool` type supports variables with the values `true` and `false`. A conditional expression may be an arithmetic expression, a `bool` expression or a pointer expression.

The `if` statement is a language construct which switches the program flow depending on the evaluation of a conditional expression. When true the statement following the `if` is executed, else the statement following the `else` is executed. Note that the `else` clause is optional and that when several statements need to be executed a compound block (braces) may be used.

The if Statement

note declaration
anywhere in block

```
#include <iostream>

int main()
{
    std::cout << "Input a number greater than 10";
    int number;
    std::cin >> number;
    if (number > 10)
        std::cout << "Correct, thank you";
    else
        std::cout << "Sorry, you're fired!";
}
```

Indentation emphasises code structure and makes the program easier to read

The language is free-form and does not depend on layout

The conditional expression evaluated by the `if` construct must be enclosed in brackets. This expression may be made up of several components, or even a single integer value.

The above example also demonstrates that variables may be declared anywhere within the function before they are used. This was not the case for the C language which insisted that variable declarations appear at the top of a function block.

It is considered good programming practice to indent program text to make statement relationships explicit. All statements inside `main()` are indented to show they belong to `main()`; the statements within the `if` construct are indented to show they related to it.

A Looping Statement

- The `while` statement iterates while true

```
while (conditional expression)
    statement;
```

- The statement may be a compound or block

```
while (conditional expression)
{
    statement;
    statement;
    ...
}
```

Unlike the `if` statement, which simply switches program flow, the `while` statement iterates around a series of statements while a condition remains true. As soon as the condition changes, the loop is terminated.

Sometimes it is useful to iterate around several statements (or in the case of the `if` statement to have several statements in each branch of the program). This can be achieved by using a compound statement or block. The block consists of an opening brace, a sequence of statements, and then a closing brace. Blocks may appear anywhere a statement can appear.

The While Statement

```
#include <iostream>

int main()
{
    std::cout << "Input a number greater than 10: ";
    int number;
    std::cin >> number;

    // a C++ comment
    while (number < 10)
    {
        std::cout << "Incorrect, please try again: ";
        std::cin >> number;
    }
    std::cout << "Correct, thank you" << std::endl;
}
```

```
Input a number greater than 10: 5
Incorrect, please try again: 6
Incorrect, please try again: 11
Correct, thank you
```

In the above example the loop iterates until the user inputs a value greater than 10. Once this occurs the conditional statement evaluates to false and the loop terminates.

Notice in the final cout statement the use of the end-of-line manipulator. endl looks like a normal variable but is in fact a request telling cout to output a new line. Otherwise the cursor remains flashing after the word 'you', rather than the next line.

C++ Comments

C++ has two mechanisms for comments. The first is inherited from C and involves enclosing the commented area within two tokens

```
/*
    this is all comment now until
    eventually the terminating
    comment symbol is found
*/
```

The problem with this form of comment is that it does not nest properly. If a second comment is placed within the first (accidentally), then when this ends it will prematurely terminate the first.

In C++ the preferred comment is '//'. This comments from the point of use to the end of the current line.

```
// this is a comment
c = c + 1;    // here is some code
// here is another comment.
```

You can comment /* and */ using //, and you can comment // within /* and */.

A Function

- **Programmers write functions to**

- structure the program, reduce replication
- focus attention on what the function does, not how it is done
- promote re-use since libraries of functions may be shared

```
void functionName()  
{  
    statement  
    ...  
}
```

A void function carries out some work but does not return a value when complete

```
int functionName()  
{  
    statement  
    statement  
    return  
}
```

An int function carries out some work and returns an integer when complete

An important concept in most high-level languages is the ability to divide the program into a series of distinct functions. This improves the modularity and structure of the code, promoting re-use, ease of modification and renders the program easier to read.

There are principally two kinds of functions in C++, those which return a value and can therefore be used in expressions, and those which do not. Functions which return values must be declared with a type which represents the type of value they return. Those which do not return values, are declared as `void`. In other languages, functions are referred to as procedures or sub-routines.

Better Facilities in C++

In addition to the conventional use of functions, C++ allows functions to be integrated into data structures. Such structures are referred to as objects and enable concepts to be modelled by encapsulating all relevant state data and functions.

A Function

```
#include <iostream>

int getnum()
{
    int number;
    std::cout << "Please enter a number: ";
    std::cin >> number;
    return number;
}

int main()
{
    int a, b;
    a = getnum();
    b = getnum();

    if (a > b)
        std::cout << a << " is the largest";
}
```

The diagram shows two horizontal arrows pointing from the code to the right. The first arrow originates from the `return number;` line in the `getnum()` function and points to the text "This value is returned to the caller". The second arrow originates from the `a = getnum();` line in the `main()` function and points to the text "Calls getnum(), and assigns result to a".

The program consists of two functions. `getnum()` prompts the user for an integer number and then returns this to the caller. The function is called twice in `main()`, to read values for variables `a` and `b`.

Note that in this example it was necessary to define `getnum()` before `main()`. This is because C++ requires all functions (just as with variables) to be declared before use. Forward declaration of functions is the usual means of achieving this and is discussed later in the course.

The example demonstrates one of the advantages of using functions. The code in `getnum()` only appears once in the program but is used twice. Consequently the program is smaller, and since no human effort was required in duplicating the code, there is less chance of error. However, there is an overhead in functions, but this should only be considered when writing very small functions.

C++ I/O Stream Objects

In the above example, `cin` and `cout` are objects of `istream` classes. In addition to the high-level operations performed with `>>` and `<<`, the objects may also be used to perform low-level I/O.

```
char c;
cin.get(c)    // reads the next character into c
cout.put(c)   // writes the character in c
```

Class member functions are presented later in this course, and the complete family of I/O Stream classes in the C++ Programming course.

Compiling C++ Code

- **Stages of compilation**

- pre-processing
- compiling
- inline expansion
- template instantiation
- code analysis, optimisation & generation
- link editing

.C .cc .cpp	source file
.i	pre-processor output
.s .S	assembler source
.o	object file
.a	static library for linker
.so	dynamic library for linker

C++ programs must be pre-compiled before they can be executed. The process consists of several stages of translation, and results in an executable program which may be run directly.

The pre-processor stage involves expanding macros and include files. The `#include <iostream>` command is a directive to the pre-processor requesting that the `iostream` file is read into the program. The path for the file is determined by compiler defaults, the environment, or command line options (`-I`).

Compilation is the main stage of the compilation process in which the C++ code is transformed into an object code. This is a form suitable for the underlying system architecture with symbolic information. The latter identifies functions and variables which are accessed but not defined in the program. The information is used by the linker. This links the object file of the program to other object files and libraries to resolve any undefined symbols.

The actual compilation sequence, and indeed the number and type of steps depends on the compiler system being used.

How to Write & Compile a Program

- Use any text editor to generate the program file

```
$ vi myprog.C
```

xemacs, slickedit, vim

- Invoke the C++ compiler

```
$ CC myprog.C
```

popular compilers include
Sun Studio, GNU g++, Comeau

- Execute the program

```
$ ./a.out
```

default output file name
a.out

Writing and compiling a C++ program is quite straightforward, but will depend on the underlying environment. You may use integrated development environments (such as Visual Studio or Eclipse IDE), or work from the command line using a variety of tools.

Use `vi` or similar to construct the program file, and then use the C++ compiler to generate the output file. Some C++ compilers are invoked with the command `CC`, the GNU compiler uses `g++`. The Comeau compiler is a standards compliant reference compiler that compiles to intermediate code. It is useful for test compiles to help spot bugs in production compilers.

By default compiled programs on Unix™ machines are given the name `a.out`. This may be changed by using the `-o` option with the compiler. To execute the program, type its name (relative names must be reachable from the shell `PATH`).

Standard Compiler Options

- Important compiler options

```
$ CC [options] [source-files] [object-files] [libraries]
```

-V	print compiler version number
-c	compile only, producing a .o for linking
-o	set name of output executable
-g	prepare file for debugging
-xO?	Control optimiser, -xO5 highest
-l	add library to linker's list
-L<path>	add path to the library search path
-I<path>	add path to pre-processor include path

```
$ CC a.C b.C c.C -o prog
$ ./prog
```

```
$ CC -g a.C b.C c.C
$ CC -XO5 fast.C
```

```
$ CC -c a.C b.C c.C -lmylib -L$HOME/myLibs -I/$HOME/myHeaders
$ CC a.o b.o c.o d.C
$ ./a.out
```

Writing and compiling a C++ program is quite straightforward, but will depend on the underlying environment. You may use integrated development environments (such as Visual Studio or Eclipse IDE), or work from the command line using a variety of tools.

Use `vi` or similar to construct the program file, and then use the C++ compiler to generate the output file. Some C++ compilers are invoked with the command `CC`, the GNU compiler uses `g++`. The Comeau compiler is a standards compliant reference compiler that compiles to intermediate code. It is useful for test compiles to help spot bugs in production compilers.

By default compiled programs on Unix™ machines are given the name `a.out`. This may be changed by using the `-o` option with the compiler. To execute the program, type its name (relative names must be reachable from the shell `PATH`).

Course Notes



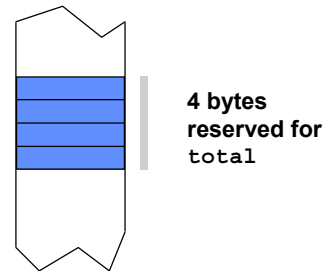
Chapter 3: Intrinsic Data Types

- **Types & Variables**
- **Fundamental Types**
- **Standard Library Types**
- **Using `string`**
- **`const` and `volatile` Type Qualifiers**
- **Pointer Types**
- **Reference Types**
- **Enumeration Types**
- **Array Types**
- **`typedef`**

Types and Variables

- Type refers to the organisation of memory
- A variable is an allocated chunk of memory
- All variables in C++ must have a type
- Most variables have an associated name (identifier)

```
int main()
{
    int total;
}
```



In C++ all variables must have a type. The type indicates to the compiler how much memory should be reserved for the variable, and indicates how the memory should be structured. In addition, it suggests what operations are valid on the variable.

In C++ variables are also referred to as objects, and types are referred to as classes. In object-oriented systems the class (or type) indicates not only what data is associated with the object, but also the operations which may be applied to it.

In the example above the variable `total` is introduced into the program with a type of `int`. On most systems, this will cause 4 bytes of memory to be allocated and associated with the name `total`.

Fundamental Types

- C++ has a set of fundamental types

- To represent integers of different sizes

```
bool
char
wchar_t
short int
int
long int
long long
```

`wchar_t` for wide characters

unsigned char | int | long
to use sign bit for number

- To represent floating point numbers

```
float
double
long double
```

Note that floating point
numbers require a
mantissa and exponent

The various types to represent integer and floating point numbers differ in the number of bytes allocated. The language (like C) does not specify exactly how big each should but leaves this up to individual implementations. However, the following are usual on Sun machines:

on the side

The number of bytes associated with each type depends on the underlying hardware. The language specified relative sizes

<code>char</code>	1 byte
<code>wchar_t</code>	2 bytes
<code>short int</code>	2 bytes
<code>int</code>	4 bytes
<code>long int</code>	4 bytes
<code>long long</code>	8 bytes
<code>float</code>	4 bytes
<code>double</code>	8 bytes
<code>long double</code>	16 bytes

The rule presented by Stroustrup in his Annotated Reference Manual (ARM) is that `short` may be no bigger than an `int`, and `long` is no smaller than an `int`.

The floats tend to be bigger since they must accommodate both exponent and mantissa.

In addition to these, unsigned integer types provide storage for positive numbers (but twice as big since the sign bit is available).

Characters are handled in C++ as one byte integers (or two bytes in the case of `wchar_t`). Whatever the character set associated with the machine (ASCII or EBCDIC), the corresponding integer is stored. When reading or writing `char`'s the actual character is expected to be input and will be displayed.

Standard Library Types

- The standard library adds many useful types

- scalar data types

<code>string</code>	sequence of characters
<code>complex</code>	numeric type for complex numbers

- collection data types

<code>vector</code>	sequence, efficient random access
<code>bitset</code>	container for storing bits
<code>deque</code>	double ended queue (deck)
<code>list</code>	linear sequence, efficient insertions
<code>set</code>	set of unique keys
<code>multiset</code>	set with non-unique keys (bag)
<code>map</code>	map of unique key to values
<code>multimap</code>	map of non-unique keys to value

Using string

```
#include <iostream>
#include <string>

int main()
{
    std::cout << "First name: ";
    std::string first;
    std::cin >> first;
    std::cout << "Second name: ";
    std::string second;
    std::cin >> second;
    std::cout << "hello " << first + " " + second << std::endl;
}
```

```
First name: dave
Second name: mallon
hello dave mallon
```

+ operator for string
concatenation

In the example three variables are first declared. Their values after the declaration are undefined. Next, the variables are assigned literal values appropriate to their types. Although their values are now defined, there was a period during the program in which the values were unknown. To avoid this dangerous state, variables may be initialised when they are declared.

The `hexnum` variable is initialised with a value when it is declared. The value is a hexadecimal integer literal.

The variable `a` is initialised with the character literal `'A'`. There may seem to be a type conflict here. However, since characters are integers (albeit small ones) they may be stored in `int`'s. When the content of `a` is displayed it shows the integer value of the character `A` in the machines local character set. In this example, this is ASCII.

Variable Names

- **The name of a variable may be composed of**
 - letters, digits, the underscore character
 - upper / lower case is significant
- **Limitations**
 - first character may not be a digit
 - may be reserved if starts with two or more underscores
 - compiler may restrict the length
 - language keywords cannot be used
- **Conventions**
 - usually written in lower case
 - choose identifier with meaning in the program

<code>is_empty</code>	or	<code>isEmpty</code>
-----------------------	----	----------------------

Variable names are identifiers used to represent allocated, typed, memory. In addition to not having a leading digit, variables may not be reserved words.

asm	auto	break	case	catch
char	class	const	continue	default
delete	do	double	else	enum
extern	float	for	friend	goto
if	inline	int	long	new
operator	private	protected	public	register
return	short	signed	sizeof	static
struct	switch	template	this	throw
try	typedef	union	unsigned	virtual
void	volatile	while	bool	true
dynamic_cast	const_cast	reinterpret_cast	static_cast	false
explicit	mutable			

Variable Names

- Which of the following identifiers are invalid?

```
int main()
{
    int __ident;
    int alpha1, alpha2;
    double salary;
    int 123;
    int salary;
    short int A;
    long double while;
    int ALPHA1;
}
```

**Beware reserved words,
redeclarations, leading
digits and underscores**

Variables may not be redeclared within the same lexical scope. For the moment, this simply means within the same block.

In the above, several of the identifiers would cause compilation errors.

```
int __ident;           // dangerous
double salary;         // multi-declaration
int 123;               // starts with digit
int salary;            // multi-declaration
long double while;     // reserved word
```

Namespaces

- **Control the visibility of names**
 - packaging classes, objects & functions under a single name
 - reduces global namespace pollution

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    cout << "First name: ";
    string first;
    cin >> first;
    cout << "Second name: ";
    string second;
    cin >> second;
    cout << "hello " << first + " " + second << endl;
}
```

In the example three variables are first declared. Their values after the declaration are undefined. Next, the variables are assigned literal values appropriate to their types. Although their values are now defined, there was a period during the program in which the values were unknown. To avoid this dangerous state, variables may be initialised when they are declared.

The `hexnum` variable is initialised with a value when it is declared. The value is a hexadecimal integer literal.

The variable `a` is initialised with the character literal `'A'`. There may seem to be a type conflict here. However, since characters are integers (albeit small ones) they may be stored in `int`'s. When the content of `a` is displayed it shows the integer value of the character `A` in the machines local character set. In this example, this is ASCII.

Literal Constants

- Literal constants are actual values and are typed

```
1          // int
1L         // long
3.14      // double
'a'       // char
```

```
200        // decimal
023        // octal
0xff       // hexadecimal
```

```
'a'        // character a
'2'        // character 2
' '        // blank
\n         // newline
\a         // bell
\0         // null
\\         // backslash
```

Literal constants are actual values used in the program. When a variable is assigned or initialised with the value 85, this is a literal constant. The value is handled by the compiler and placed in memory when the program is executed.

Like most things in C++, literals are typed, so there are int literals and char literals. Although C++ does not support strings as a fundamental type, there are also string literals. These are arrays of characters and discussed later in this chapter.

Literals may be qualified with l (or L) and u (or U) to indicate long and unsigned respectively.

Types, Variables & Literals

```
#include <iostream>

int main()
{
    int i;
    char c;
    double d;

    i = 33;
    c = 'a';
    d = 3.14983;

    int hexnum = 0x522;
    float f = 332.2;

    int a = 'A';
    std::cout << "value in a: " << a;
}
```

Declarations of three variables

Assigning literal values to each variable

Declaring and initialising two variables

value in a: 65

In the example three variables are first declared. Their values after the declaration are undefined. Next, the variables are assigned literal values appropriate to their types. Although their values are now defined, there was a period during the program in which the values were unknown. To avoid this dangerous state, variables may be initialised when they are declared.

The `hexnum` variable is initialised with a value when it is declared. The value is a hexadecimal integer literal.

The variable `a` is initialised with the character literal `'A'`. There may seem to be a type conflict here. However, since characters are integers (albeit small ones) they may be stored in `int`'s. When the content of `a` is displayed it shows the integer value of the character `A` in the machines local character set. In this example, this is ASCII.

const and volatile Type Qualifiers

- **Constants are variables whose values never change**

- must be initialised; cannot be assigned

```
const int bufSize = 512;  
const char code = 'U';
```

- **Constants should be used whenever possible**

- more meaningful than literals; safer when value shouldn't change

```
bufSize = 10;
```

Unless bufSize is const, this error may go unnoticed.

- **volatile type qualifier**

- prevents variable optimisation as value may change unexpectedly

```
volatile const int time = 1280774720;
```

All types may be qualified with the keyword `const`. This means that variables of the type cannot be changed once initialised.

The notion of `const` is useful in programming because it enables the compiler to verify that the intention (that the variable should not be changed) is indeed being adhered to.

C++ makes considerable use of `const`. Because of the relationship between types (the conversions which may take place) it is important that `const` is used whenever possible. Failure to do so may restrict a program's usefulness.

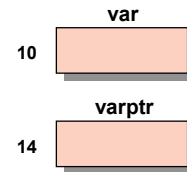
volatile

The `volatile` qualifier indicates that a variable may have its value changed in ways outside the control of the compiler (by the operating environment, a system clock or another program). The qualifier stops the compiler optimizing code referring to the variable by storing the variable's value in a register (or read-only memory if `volatile const`) and re-reading it from there. Instead, the compiler must generate code to re-read it from memory, where it may have changed.

Pointer Types

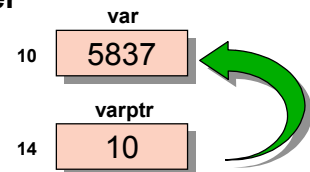
- A pointer is a variable that may contain the address of a variable

```
int var;           // variable
int *varptr;       // pointer
```



- Assigning an address to the pointer

```
var = 5837;
varptr = &var;
```




C++ has inherited from C a comprehensive pointer system. Both C and C++ allow relatively low-level manipulation of data, and highly flexible construction of data structures. A basic requirement for this is the ability to deal directly with a programs memory.

A pointer is a integral variable which is typed such that it may only hold the address of other variables. C++ provides operators to take the address of a variable so that it may be assigned to a pointer. It also provides an operator to dereference a pointer, so that the data which it points to may be accessed.

Pointer Types

- **Pointers are important in C++**
 - allow multiple references to one memory location
 - allow custody of objects / variables to be passed around
 - often used with linked-list and dynamic memory allocation
- **Important operators for pointers**



 - indirection operator, allows access through pointer
 - address operator, takes the address of a variable
- **All pointers have an associated type**
 - an integer pointer may not hold the address of a float variable

Pointer Types

```
#include <iostream>

int main()
{
    int total;
    std::cout << "enter total: ";
    std::cin >> total;

    int *total_ptr = &total;
    std::cout << total << *total_ptr;
    *total_ptr = *total_ptr + 1;
    std::cout << *total_ptr << total;
}
```

total_ptr	&total
total *total_ptr	51

```
enter total: 50
50 50
51 51
```

on the side

Note that a pointer is simply a variable that can hold the address of another. If it does not hold a valid address, then it should not be dereferenced.

In this example the variable `total` is assigned the value 50 (input from the keyboard). `total_ptr` holds the address of `total`. When `total_ptr` is dereferenced it provides access to `total`. Changes to `*total_ptr` are the same as those to `total`, because the same part of memory is being changed.

Note the distinction between declaring a pointer variable and using a pointer variable.

```
int *total_ptr
```

This introduces a variable of type pointer-to-int.

```
*total_ptr
```

Applies the indirection operator to `total_ptr` to access whatever it points at.

Reference Types

- **Automatic pointer mechanism**
 - provides ability to make aliases to variables
 - less flexible than general pointers
 - & and * operators are no longer required
 - especially useful for function arguments

```
#include <iostream>

int main()
{
    int i = 5;
    int &iref = i;

    std::cout << i << iref;
    i = i + 1;
    std::cout << i << iref;
    iref = iref + 1;
    std::cout << i << iref;
}
```

**Note: references
must be initialised**



5 5
6 6
7 7

C++ provides another mechanism similar to pointers. References are a form of automatic pointers which provide aliases to variables. Once a reference has been established the variable may be accessed through its original name or through the reference, both are identical.

Unlike pointers, it is not necessary to take the address of the variable being referenced (&) or to apply the dereference operator to the reference (*) to see what it points to.

References provide a mechanism of passing variable parameters into functions, and enable functions to (easily) appear on the left hand side of an expression. The significance of this will become apparent when overloading operators.

Enumeration Types

- An enumeration type is a user-defined set of values
 - symbolic integral (integer) constants
 - enumerators (the symbols) exist only at compile time
 - often more meaningful and convenient than using literals

```
enum Choice { No, Yes };
```

0 1

on the side

Since the standard now includes a `bool` type, this particular enum is of course redundant!

```
enum Choice { No, Yes };

Choice value = Yes;
while (value)
{
    std::cout << "going forever";
}
```

going forever
going forever
going forever
...

An enumeration type is a user-defined list of symbolic constants representing integral values. Enumerations provide a convenient way to associate constant values with names. In the above example, the programmer may think in terms of `Yes` and `No` rather than 1 or 0. When a complicated list of alternatives exist, the symbolic names convey more information than integer codes.

An Enumeration defines a type which is a sub-set of `int`. By default, the values specified in the enumeration list are associated with values starting from the left with 0. Consequently, enumeration values may be used in place of `int`'s. The opposite is not true however. It is not possible to assign a variable of an enumeration type an integer value. Such variables may only be assigned symbolic values as specified in the enumeration list.

```
enum Choice { No, Yes };

Choice var1 = Yes;           // enum value Yes
// Choice var2 = 1;          // invalid for type
int var3 = 1;                // just an integer
int var4 = Yes;              // fine, Yes is interpreted as 1
```

Unlike constants, enumerations are compile time constants and may not ultimately take up run-time memory.

Enumeration Types

- **enums define symbolic names for integer values**

```
enum Lights { RED, YELLOW, AMBER = YELLOW, GREEN };  
  
enum { OPEN, CLOSE, READ, WRITE, CREATE };  
  
enum { BUFSIZE = 1024 };
```

- **enums may have an anonymous type**

```
enum { No, Yes };  
  
while (Yes)  
{  
    std::cout << "going forever";  
}
```

Anonymous enumerations simply introduce symbolic constants into the program for use with integer variables. Since the enumeration has no type name, it is impossible to make variables specifically of the enumeration type.

The above examples demonstrate that it is also possible to define the integral value which an enumeration should have, or indeed to make two symbols have the same value.

Array Types

- **An array is a collection of objects of one type**
 - individuals do not have identifiers, are referenced by index

```
int i;           // declares 1 integer
int ai [10];     // declares 10 integers
```

- **Array length is specified when declared**
 - elements range from 0 to length - 1
 - [] (subscript) operator is used to index an array

```
ai [0] = 1;
ai [2] = 4;
if ( ai[9] > 44 )
....
// ai [10]
```

danger, subscript
out of range

Arrays offer a mechanism of referring to a collection of related values from one identifier. This allows the values to be manipulated within the program as a single unit.

Accessing individual elements within an array is achieved using the subscript operator []. This takes a single integer operand to indicate which element is of interest. Once indexed, elements of an array type behave just like conventional variables.

In C++, arrays must be declared with the subscript symbol. This usually contains a number indicating the length of the array. Elements may then be accessed from zero to length minus one.

Intrinsic arrays provide no run-time query to determine length---unlike the standard template libraries counterpart.

Array Types

- **Arrays are useful for keeping related data together**
 - array of integers (a vector)
- **Arrays can be used with user-defined types**
 - array of employees
- **Arrays can be initialised using an initialisation list**

```
int vector [5] = { 1, 2, 3, 4, 5 };  
float angles [] = { 3.2, 4.2, 4.3 };  
char name [] = { 'C', '+', '+' };
```

- **Array is auto-sized when the size is omitted**

Just as single variables can be initialised when they are declared, array variables may also be initialised. Since arrays contain several values (for each of their elements) a list of initial values must be provided. The list comprises a set of comma separated values (of the appropriate type) delimited by braces.

When initialisation lists are provided it is not necessary for the size of the array to be explicitly stated. The compiler can determine what size it should be from the number of elements in the list. If a size is provided then take care to ensure this is consistent with the list. If the array is larger than the list then the extra elements are left undefined. It is an error to make the array smaller than the number of initialisers.

C++ supports multi-dimensional arrays by repeated application of the subscript operator.

```
int grid [5][5];    // declares a 5 x 5 array  
grid [0][0] = 3;    // indexes element 0,0
```

In C++, like C, arrays are intimately linked to pointers. In fact, the two can often be used interchangeably. In the example on the foil, `vector` is considered to be a pointer-to-int. `*vector` would realise the value 1.

Character Arrays

- C style strings implemented as arrays of characters

```
char name[] = { 'C', '+', '+' };  
char computer[] = "apple";
```

name [] C + +

computer [] a p p l e \0

- Double-quotation marks indicate a string constant
 - arrays may be initialised with string constants
 - all string constants are delimited by a null character
- Always use the `string` library in production code
 - often implemented using `char *`, and is easier to use

Above, `name []` has only three elements, while `computer []` has six. The reason for this is that `computer []` was initialised from a string constant. String constants are defined by placing a set of characters between double-quotation marks. All string constants are automatically padded with the null character. This enables the end of string to be detected when functions are manipulating the character elements.

Array Types

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    char rev[] = "reverse";
```

```
    int i = 0;
```

```
    while (rev[i] != '\0')
```

```
        i = i + 1;
```

```
    std::cout << "No of chars: " << i;
```

```
    while (i >= 0)
```

```
    {
```

```
        std::cout << rev[i];
```

```
        i = i - 1;
```

```
    }
```

```
}
```

while rev at index i
does not equal end of
string

output contents of rev
in reverse order

```
No of chars: 7  
esrever
```

The example introduces two new relational operators

`!=` 'not equal' is true when its operands are different

`>=` 'greater or equal' is true when left is `>=` than right

In the program the first while loop is used to iterate the variable `i` through the list of characters in the array. The loop terminates when the null (end of string) character is detected. The value of `i` is displayed as the length of the string.

The second loop again iterates through the elements of the array of characters. However, the value of `i` now controls the loop. This is progressively decremented and the loop terminates when it is zero. As `i` is decremented the corresponding character in the array is printed so that in the end its contents are written in reverse.

Another way of writing the second loop is to remove the relational test.

```
while (i)
{
    std::cout << rev[i];
    i = i - 1;
}
```

This succeeds because when `i` is decremented to zero, then the loop terminates. Recall, that conditional statements are zero or non-zero, false or true.

There is, however, a flaw in the above code fragment. The loop actually terminates before `rev[0]` is written. An exit condition loop would resolve this problem.

Array Types

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    char rev[] = "reverse";
```

```
    int i = 0;
```

```
    while (rev[i] != '\0')
```

```
        i = i + 1;
```

```
    std::cout << "No of chars: " << i;
```

```
    while (i >= 0)
```

```
    {
```

```
        std::cout << rev[i];
```

```
        i = i - 1;
```

```
    }
```

```
}
```

while rev at index i
does not equal end of
string

output contents of rev
in reverse order

```
No of chars: 7  
esrever
```

The example introduces two new relational operators

`!=` 'not equal' is true when its operands are different

`>=` 'greater or equal' is true when left is `>=` than right

In the program the first while loop is used to iterate the variable `i` through the list of characters in the array. The loop terminates when the null (end of string) character is detected. The value of `i` is displayed as the length of the string.

The second loop again iterates through the elements of the array of characters. However, the value of `i` now controls the loop. This is progressively decremented and the loop terminates when it is zero. As `i` is decremented the corresponding character in the array is printed so that in the end its contents are written in reverse.

Another way of writing the second loop is to remove the relational test.

```
while (i)
{
    std::cout << rev[i];
    i = i - 1;
}
```

This succeeds because when `i` is decremented to zero, then the loop terminates. Recall, that conditional statements are zero or non-zero, false or true.

There is, however, a flaw in the above code fragment. The loop actually terminates before `rev[0]` is written. An exit condition loop would resolve this problem.

typedef

- Create an alias for an existing data type
 - useful if the type is unwieldy

```
int main()
{
    typedef unsigned int Ui;
    Ui bignum;

    typedef int Arri [100];
    Arri numbers;

    typedef char *Arrcp [100];
    Arrcp environ;
}
```

syntax the same as would be used to
declare a variable of this type

A useful short hand notation that is extremely useful when the types being created are complex. For example, where one type nests inside another, or when using iterative type definitions. Pointer types can be particularly tricky sometimes, and typedef can simplify this.

Some advanced uses of typedef occur with templates to resolve maximal munch lexical issues; and also with generic programming to support the trait mechanism of externalising internal types.

Course Notes

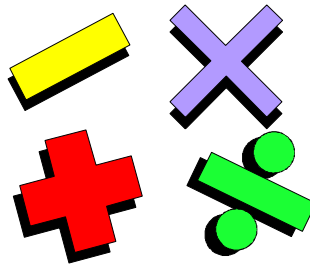


Chapter 4: Operators & Expressions

- **Arithmetic Operators**
 - **Relational & Logical Operators**
 - **Assignment Operators**
 - **Precedence & Associativity**
 - **Increment & Decrement Operators**
 - **Comma Operator**
 - **Bitwise Operators**
 - **Conversion Operators**
-

Operators & Expressions

- Operators are used in expressions to modify variables



arithmetic
relational
logical
bitwise
conversion

unary
binary
ternary

Operators are used in expressions to modify and test variables and literals (which may themselves be generated from other expressions).

Binary operators take two operands (or arguments), unary operators take one operand. C++ also has one ternary operator, the 'arithmetic if' operator to be discussed in the next chapter. Note that the number of operands associated with an operator is referred to as its arity.

Arithmetic Operators

- C++ supports the usual set of arithmetic operators

+	unary plus	+expr
-	unary minus	-expr
*	multiplication	expr * expr
/	division	expr / expr
%	modulus (remainder)	expr % expr
+	addition	expr + expr
-	subtraction	expr - expr

Operators within the same group have the same precedence. Operators in a higher group have a higher precedence

- **Division & Modulus**

- fractional parts are discarded during integer division
- modulus computes the remainder for integer division

All the usual arithmetic operators are supported in C++. The behaviour of the operators depends on the operands to which they are applied. In integer arithmetic, the division operator discards any fractional part; in floating point arithmetic this is maintained. The modulus operator is only used in integer arithmetic; it divides the operands and evaluates to the remainder. A common use is to normalise data within a defined range of values. See example later in this chapter.

Unary minus changes the sign bit of its operand. The operator is undefined for unsigned data types. Unary plus is superfluous and was introduced to keep minus company.

Relational & Logical Operators

- **Relational and logical operators evaluate to true or false, that is, non-zero or zero**

!	logical NOT	!expr
<	less than	expr < expr
<=	less than or equal	expr <= expr
>	greater than	expr > expr
>=	greater than or equal	expr >= expr
==	equality	expr == expr
!=	inequality	expr != expr
&&	logical AND	expr && expr
 	logical OR	expr expr

The relational and logical operators evaluate to true (>0) or false (0) depending on their operands. Logical AND is true if both operands are true, logical OR is true if either operand is true. Logical NOT is true if its operand is false, and vice versa.

The equality and inequality operators are true if their operands have the same value or don't have the same value respectively. Note that the equality symbol consists of two characters '==', and should not be confused with the assignment operator.

Relational & Logical Operators

```
int main ()
{
    if (ready = 1) ...
    if (ready == 1) ...
    if (!found) ...
}
```

Beware! assignment not equality test

if not found ...

- C++ employs lazy (short circuit) evaluation

```
int main ()
{
    if (found && ptr != 0 && *ptr == 'G')
        ...
}
```

Execution stops as soon as failure detected,
leaving other parts unevaluated

The first example illustrates the problem with the assignment and equality operators, they are too similar. By using assignment above, the value in the `ready` variable will be changed to 1, and the expression inside the `if` statement considered true. A common programming practice to avoid this problem, is to put the constant variable or literal on the left-hand side of the test.

```
if (1 = ready) ...
```

This produces a compile time error since 1 is not an l-value, thus causing the problem to be found.

An important rule in C++ and C is that both languages employ lazy evaluation when interpreting logical expressions. In the second example above, three conditions must be true in order for the expression to evaluate to true. C++ interprets the expression from left to right, as soon as one condition fails, the whole expression fails. Other parts of the expression are left untouched.

The logical AND operator (&&) effectively guards against incorrect expression evaluation. If `found` is false, then the `ptr` is not examined or dereferenced. If `found` is true then the pointer is examined. Only if it contains an address is it subsequently dereferenced. This is important, because dereferencing an invalid pointer in C++ causes a run-time error.

Assignment Operators

- During assignment a value is stored in the left operand

=	assignment	lvalue = expr
---	------------	---------------

- **l-values & r-values**

- l-value refers to the address in memory of a variable
- r-value refers to the value stored in a variable

```
total = num;  
num = total;  
  
num = 3;  
// 3 = num;
```

variables have both l-value and r-value components, and may appear on either side of an assignment

literal constants are r-values, they cannot be l-values because they have no associated memory

The assignment operator takes the value from the right hand operand and stores this in the location associated with the left hand operand. In order to succeed, it is necessary that the left operand has a location associated with it. Since literal constants do not have such space, they may not appear on the left side of an assignment.

Variables associated with memory are called l-values because they may appear on the left of an assignment; both variables and literals are r-values. Note that although constant variables have an associated memory location, the C++ type rules prohibit them from being lvalues; by definition their values may not be changed.

Assignment Operators

- A family of assignment operators simplify self-assignment expressions

<code>+=</code>	plus equals	<code>expr += expr</code>
-----------------	-------------	---------------------------

```
total = total + column;  
total += column;  
  
average = average / sum;  
average /= sum;
```

- Other members of the family

<code>-=</code>	<code>*=</code>	<code>/=</code>	<code>%=</code>	<code>&=</code>	<code>^=</code>	<code> =</code>	<code><<=</code>	<code>>>=</code>
-----------------	-----------------	-----------------	-----------------	---------------------	-----------------	-----------------	------------------------	------------------------

There are a family of assignment operators which make expressions more compact and often clearer. These compound assignment operators are useful when an operand is being operated upon and then assigned to itself.

Note that the operators are distinct from their counterparts. Each operand is taken as a single token in the language, '`>>=`' is a single operator token and is distinct from either '`>>`' or '`'=`'.

Increment & Decrement Operators

- Enable 1 to be added or subtracted easily

++	post-increment	lvalue++
++	pre-increment	++lvalue
--	post-decrement	lvalue--
--	pre-decrement	--lvalue

- The operand must be an l-value

```
sum = sum + 1;  
sum += 1;  
++ sum;
```

All these statements
produce the same result,
sum is incremented by 1

A common operation in C++ is to add or subtract 1 from a variable. Two operators are provided for this purpose, increment and decrement. The operators may only be applied to lvalues.

Increment & Decrement Operators

- **Pre-increment**
 - object is incremented and expression evaluates to result
- **Post-increment**
 - object is incremented but expression evaluates to old value

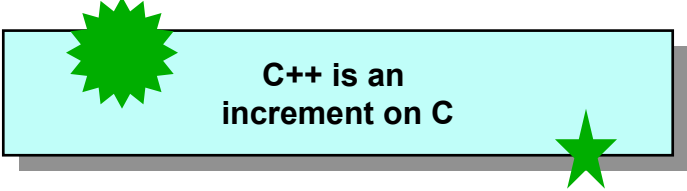
```
#include <iostream>

int main()
{
    int i = 5;
    std::cout << "i is " << i << std::endl;
    std::cout << "i++ is " << i++ << std::endl;
    std::cout << "i is " << i << std::endl;
    std::cout << "++i is " << ++i << std::endl;
}
```

i is 5
i++ is 5
i is 6
++i is 7

Pre- increment and decrement change the value of the operand and as an expression evaluate to this new value. Post- increment and decrement change the value of the operand but as an expression evaluate to the old value of the object.

In the second cout of the example, i is incremented but this is not apparent until i is output on the next line.



C++ is an
increment on C

Increment, Precedence & Modulus

```
#include <iostream>

int main()
{
    int i = 0;
    while (1)
        std::cout << i++ % 10 << " ";
}
```

Precedence order:

++ % <<

Post-increment:

i starts with 0

Modulus:

normalises i for 0 .. 9

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
```

The loop is infinite, since a literal cannot be changed and 1 is true. The only way to leave such loops is to 'break' out of them as discussed in the chapter on looping constructs.

The expression involves three operators and is evaluated according to their precedence. The post-increment is applied to *i*, its previous value is divided by 10 and the remainder (the modulus) held as the result, this is then output to the display followed by a space. The reason the space isn't output first is that '<<' is left associative.

The example demonstrates some of the utility of the modulus operator. Here it is used to normalise the value in *i* within a specified range.

Comma Operator

- **Allows a series of expressions to be treated as one**
 - This operator is not often used, but can lead to elegant (sometimes unreadable) code

```
result = (++task, a = func(), average);
```

- **Often used in looping constructs**

```
#include <iostream>

int main()
{
    int i = 10;
    while (std::cout << i << std::endl, --i);
}
```

- **Left associative**

A comma expression is a series of expressions, evaluated from left to right, separated by comma's. The whole expression evaluates to the result of the last expression, the one on the right.

The comma operator tends to be used by experienced C or C++ programmers to keep code elegant and simple. The code may become very succinct, but as it tends to be used in the same scenarios, this is not usually a problem.

Bitwise Operators

- **Bitwise operators allow access to bits**
 - operands are treated as ordered collections of bits
 - operands must be of integral type
 - operators allow individual bits to be set and unset

<code>~</code>	bitwise NOT	<code>~expr</code>
<code><<</code>	left shift	<code>expr << expr</code>
<code>>></code>	right shift	<code>expr >> expr</code>
<code>&</code>	bitwise AND	<code>expr & expr</code>
<code>^</code>	bitwise XOR	<code>expr ^ expr</code>
<code> </code>	bitwise OR	<code>expr expr</code>

- **Useful for low-level systems programming**
 - setting registers and flags

Bitwise operators allow programmers to affect the state of registers and toggle bits directly in the system. Support for bitwise operations is provided by C++, and C, since both languages are applied to systems programming. Indeed, the Unix™ operating system is written largely in C.

Bitwise NOT is a complement operator, switching bits from 0 to 1 and 1 to 0. The shift operators shift the operands bits left or right, and shift in 0s in place of the lost bits. Bitwise AND and OR perform the corresponding boolean function on the operands; bitwise XOR is exclusive OR, only setting those bits which are set in either operand but not both.

Bitwise Operators

`unsigned char flag = 0xff`

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

`~flag`

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

`flag <<= 3`

1	1	1	1	1	0	0	0
---	---	---	---	---	---	---	---

`unsigned char mask = 0x31`

0	0	1	1	0	0	0	1
---	---	---	---	---	---	---	---

`flag |= mask`

1	1	1	1	1	0	0	1
---	---	---	---	---	---	---	---

```
int status ()
{
    if (accessMask & activeFlag)
        return someVal;
}
```

flag variables may contain status bits which can be examined by using various masks.

The example shows one use of bitwise operators. In this a flag is ORed with a mask variable to determine whether the corresponding bit is enabled. Sets of masks are usually defined so that different bits in the flag variable may be tested. (Note that bitfields would be a better solution to this problem).

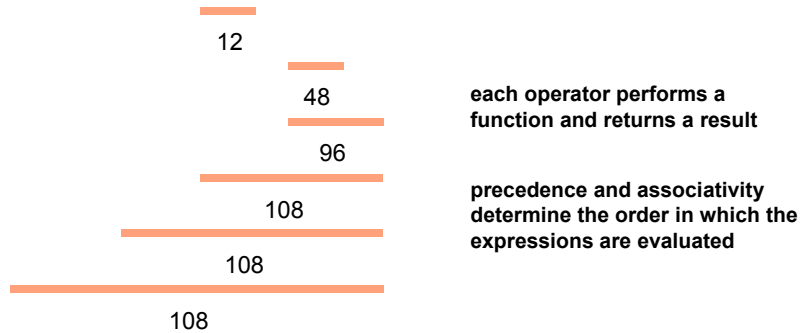
Bitwise operators are useful for systems programming and hardware manipulation.

Precedence & Associativity

- All expressions evaluate to a result

- this may be ignored
- used in subsequent expressions

```
average = total = 3 * 4 + 8 * 6 * 2
```



Most expressions are formed of multiple sub-expressions. The order in which the sub-expressions are evaluated will have an impact on the result of the whole expression. Consequently, rules of precedence and associativity guide the order in which evaluation should occur.

In the above expression, evaluation order depends partly on the compiler. Either $3 * 4$ is evaluated first, then $8 * 6$ and the result $* 2$; or $8 * 6$ and the result $* 2$; followed by $3 * 4$. Multiplication has a higher precedence than addition, so either are good. When $8 * 6 * 2$ are evaluated, $8 * 6$ is always computed first because multiplication is left associative. Assignment has a very low precedence so is executed last; it has an associativity from right to left, so results are assigned to `total`, and from `total` to `average`.

Note that in all of the operator charts shown in this chapter, groups of operators towards the top of a chart have a higher precedence than those low down. Also note that with the exception of the assignment operators and unary minus and plus, all of the operators are left associative. However, there are other operators not discussed here. It is important to understand the operator chart in the C++ programming manual.

Precedence & Associativity

- **Right Associative**

```
a = b = c = d = e = f;  
(a = (b = (c = (d = (e = f)))));
```

- **Left Associative**

```
a + b + c + d + e + f;  
((((a + b) + c) + d) + e) + f);
```

- **Highest precedence first**

```
a = b != c + d;  
(a = (b != (c + d)));
```

Right associative operators (all things being equal) are evaluated from the right; left associative from the left. Highest priority are always evaluated first.

Use parenthesis to change the evaluation order. In the above examples, the parenthesis are superfluous since the evaluation order is unchanged. Sometimes it is useful to use brackets to make a program easier to read, other times they must be used.

Use brackets to
change the order
of evaluation

```
if ((ch = next_char()) != '\n')  
{  
    ...  
}
```

In this example, `ch` is assigned a character returned from `next_char()`, and then the character (now held in `ch`) is compared with the end-of-line literal. If it is not end-of-line, the body of the `if` (a compound statement) is executed.

Sequence Points

- **A guaranteed point of expression evaluation**
 - all side-effects of sub-expression evaluations are complete
- **Sequence points are introduced**
 - for `&& !! , ?:`
 - in `if, switch, while, do-while, for` expressions
 - before a function (operator overload) is entered or returned
 - at the end of a declaration statement
- **Don't modify an lvalue more than once**

```
add = (++a)+(++a);      // unspecified behaviour
i = v[i++];             // unspecified behaviour
i = 7, i++, i++;        // i becomes 9
i = ++i + 1;            // unspecified behaviour
i = i + 1;              // the value of i is incremented
```

Under the rules for expressions evaluation in the C++ standard.

“... the order of evaluation of operands of individual operators and subexpressions of individual expressions, and the order in which side effects take place, is unspecified. Between the previous and next sequence point a scalar object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored. The requirements of this paragraph shall be met for each allowable ordering of the subexpressions of a full expression; otherwise the behavior is undefined.”

Conversion Operators

- **Conversion operators evaluate to required type**
 - conversion expression shows variable as new type
 - conversion may view the size and structure differently
 - conversions are referred to as type casts
- **C++ provides several notations for casting**

```
void foo()
{
    int i = 6;
    float f = 3.14;

    i = (int) f;           // C style
    i = int (f);           // functional
    i = static_cast<int> (f); // modern
}
```

C++ allows the contents of a variable to be viewed as a different type. Usually an anonymous variable is created and initialised with the contents of the original (though this is largely hidden from the programmer).

The process of coercing a value of one type into another is referred to as casting. When a variable is coerced into a type bigger than itself this is referred to as promotion, for example, coercing a `float` into a `double`. A coercion from a `double` to a `float`, however, is referred to as demotion. This causes a narrowing of the available memory and may cause the value to be truncated.

It is also possible to convert from `int` to `float` and vice versa. Once again, however, this may result in information loss.

C++ provides two mechanisms for casting objects of one type into another. The functional cast (not available in C) places the type as a function name and the object as an argument. The conventional C cast places the type in brackets before the object to be cast.

Conversions are very important in C++ since they allow greater flexibility in the language. They will be discussed in more detail as the course progresses.

Conversion Operators

- **Four alternate casting operators**

- less bland than C-style casts
- support compile and run-time safe type conversion
- stand out so easy to spot in the code

```
const_cast <T> (expression)
// used to remove the const or volatile of expression
// T data type must be the same as the source type

static_cast <T> (expression)
// Converts expression to the type of Target based on expression

reinterpret_cast <T> (expression)
// allows any pointer to be converted into
// any other pointer type.

dynamic_cast <T> (expression)
// casts a from one pointer or reference type to another
// performs a runtime validity check
```

The `const_cast` operator is used to cast away the constness or volatileness of an expression. The `const_cast` is a restricted cast in the sense that the compiler will only allow it to be used for this purpose. The advantage of this operator over the traditional C-style cast is that the authors intention is made clearer, and the compiler (now knowing the intention) is able to verify the code.

The `static_cast` operator is static in the sense that it is applied at compile time. It is the general purpose cast, and (with the exception of const coercions) has the same power and restrictions as the C-style cast. For example, you can't cast a struct into an int or a float into a pointer using the `static_cast`.

The `reinterpret_cast` operator changes one type into a fundamentally different type. For example, an integer into a pointer or a struct into a float. The result of a `reinterpret_cast` is likely to be implementation dependent and therefore affect the portability of code in which it is used.

Perhaps the most interesting casting operator is the `dynamic_cast`. It is dynamic in the sense that it applies at run-time to polymorphic class hierarchies. It is part of the Run-Time Type Identification (RTTI) mechanism supported in C++. The `dynamic_cast` supports down casting (actually down or cross) of a base pointer to a derived or sibling object. The cast will fail if the object referred to through the base pointer is not of the correct type. In the event of failure the operator returns a null pointer, or if the cast was applied to a reference, then an exception is thrown

static_cast

- **Similar in behaviour to C-style cast**
 - not allowed to be used for const or volatile down casts

```
#include <iostream>

int main()
{
    int number = 4;
    float result = static_cast <float> (number) / 3;
}
```

- **static_casts are evaluated at compile time**
 - able to cast to and from void *

static_casts are similar to the C style cast, though perhaps a little less powerful. The purpose of the static_cast is to generally cast from one type to another. static_casts differ from their dynamic counterparts in that they do not try to examine the objects being cast. This makes them useful for void * conversions, but useless when down casting in polymorphic hierarchies.

Implicit & Explicit Conversion

- **Conversion operators allow explicit conversions**
- **Implicit conversion may be performed by the compiler**
 - applied automatically when mixed types are used in expressions
 - follows standard conversion rules
 - rules must be defined for class types

```
int i = 3.1415      // converted to 3 implicitly
```

- **Compilers raise warnings when the data is narrowed**
 - and errors if no conversion exists

```
float f = 3 / 2;
```

Does the compiler raise a warning ?
What is the value stored in f ?

In addition to the explicit casting operators previously discussed, C++ compilers implicitly apply casts when expressions contain mixed types. Obviously this must be controlled (otherwise the type system is meaningless), and occurs according to standard conversion rules.

When implicit casts are performed the compiler raises warnings if the coercion causes a narrowing (demotion) of the data type. The warning is suppressed if the cast is explicit.

Even though the compiler may implicitly cast, an explicit cast can still be necessary. Explicit casts can make the program easier to read, and reinforce the intention of the programmer. Explicit casts may also save precision in a calculation. Consider

```
float f = 3 / 2;
```

Integer division would realise the value 1, this is coerced to a float 1.0 and assigned to f. Alternatively,

```
float f = (float) 3 / 2
```

Causes 3 to be cast to a float. C++ will always try to preserve accuracy, so the compiler will implicitly coerce 2 to a float and apply floating point division. This results 1.5 and is assigned to f.

Chapter 5: Conditional Flow Statements

- **if Statement**
 - **if-else Statement**
 - **Nesting if Statements**
 - **Arithmetic if Operator**
 - **switch Statement**
-

if Statement

- The if statement is used to make decisions

```
if (conditional expression)
    statement;
```

```
if (conditional expression)
    statement;
else
    statement;
```

- Note that statements may be blocks

```
{
    statement;
    statement;
    ...
}
```

The default flow of statement execution in C++ is sequential. Control is passed to `main()`, and then statements are executed in sequence. Conditional statements cause a branch in the program in which the sequential flow is split, based on some decision.

The fundamental statement for making decisions is `if`. This tests a condition, if the condition is true, then a corresponding statement is executed. If the condition is false, then the statement is by-passed and the program continues. In some situations two or more branches should be made. The optional `else` part of the `if` statement defines a statement to be executed if (and only if) the condition fails.

Where several statements need to be executed in each branch, a block (compound statement) may be used to encapsulate them.

if Statement

- A simple if-else statement

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

int main()
{
    char code;
    std::cout << "please enter passcode: ";
    std::cin >> code;

    if (code == 'A')
    {
        std::cout << "got code " << code << std::endl;
        func_A();
    }
    else
    {
        std::cout << "bad code " << code << std::endl;
        func_fail();
    }
}
```

The user is prompted for a passcode, which is read into the character variable `code`. If the code contains 'A' then `func_A` is invoked and a corresponding message displayed; if the code does not contain 'A' then a fail function is called and an error message displayed.

Note that the two branches are mutually exclusive. Control flow continues for both branches at the end of the `if` statement.

A common mistake for new C++ programmers is to neglect the braces which make the branches compound statements.

```
if (code == 'A')
    std::cout << "got code " << code << std::endl;
    func_A ();
else
    ...
```

The only statement associated with the `if` is `cout`. The call to `func_A` will be made whether or not `code` equals 'A'. Moreover, the `else` statement is lacking an `if`, so the compiler will raise an error.

if Statement

- A multiple if-else statement

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

int main()
{
    char code;
    std::cout << "please enter passcode: ";
    std::cin >> code;

    if (code == 'A')
        func_A();
    else if (code == 'B')
        func_B();
    else if (code == 'C')
        func_C();
    else
        func_fail();
}
```

Where multiple branches need to be made a multi-if-else statement can be devised. If code is A then `func_A()` is called and then execution continues after the entire if sequence; otherwise, the next if statement is tested. If the code is B then `func_B()` is called, after which execution continues at the end of the sequence; otherwise the next if statement is tested.

The structure of the sequence is misleading; it does not show that each if is the statement associated with the preceding else.

```
if (code == 'A')
    func_A();
else
    if (code == 'B')
        func_B();
    else
        if (code == 'C')
            func_C();
        else
            func_fail();
```

This form may be accurate, but it is somewhat less readable.

if Statement

- Nesting if-else statements

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

int main()
{
    char code;
    std::cout << "please enter two passcodes: ";
    std::cin >> code1 >> code2;

    if (code1 == 'A')
        if (code2 == 'B')
            func_AB();
    else
        std::cout << "what's happening? " << std::endl;
}
```

Beware of nesting if statements. In this example it is not clear which if statement the else belongs to.

- The else binds to the most recent if

on the side

Indent code correctly to avoid problems like this

In the above example a problem has occurred with the interpretation of the **else**. To which **if** statement does it belong? In C++ it belongs to the most recent (i.e., most nested) **if**.

The problem has arisen because of poor indentation. However, it does demonstrate that with deeply nested **if** statements (especially those subject to modification), logical errors can easily occur. The best solution is to be rigorous with indentation; or alternatively, re-structure the program without deeply nested **if**'s! (And, of course, use lot of braces).

if Statement

- Use braces to solve nested if problems

The braces
enclose the
nested if, the
else therefore
belongs to
the outer if

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

int main()
{
    char code1, code2;
    std::cout << "please enter two passcodes: ";
    std::cin >> code1 >> code2;

    if (code1 == 'A')
    {
        if (code2 == 'B')
            func_AB();
    }
    else
        std::cout << "what's happening? " << std::endl;
}
```

The 'dangling-else' problem is resolved by using braces to enclose the inner `if` statement.

Arithmetic if Operator

- The ternary if operator

```
expr1 ? expr2 : expr3
```

- Evaluates to expr2 or expr3 depending on expr1

```
if (expr1)
    expr2;
else
    expr3;
```

- Can be used just as any other expression
 - in function return statements, I/O statements, in loops

The arithmetic `if` operator is the only ternary operator in C++. Three expressions are provided, the first is the condition, the second and third are possible results. If the first expression is true, then the result is expression2, if it is false then the result is expression3.

The arithmetic `if` operator is used quite heavily in C++ as a convenient means of making decisions.

Arithmetic if Operator

- Take care with precedence when using the ternary-if

```
#include <iostream>

int main()
{
    std::cout << "enter two numbers: ";
    std: cin >> a >> b;
    std::cout << "the largest is " << (a > b ? a : b) << std::endl;
}
```

Brackets are essential because << has the highest precedence in this expression

One problem to be aware of with the ternary-if statement is precedence. The operator (? :) has a low precedence; when embedded within other expressions the operands may bind to the other operators first. To avoid this, it is common for ternary-if expressions to be enclosed in brackets.

switch Statement

- **switch is a multi-way decision making statement**
 - defines a number of branches
 - jumps to appropriate branch depending on expression

```
switch (expr)
{
    case branch1:
        statement;
        statement;
    case branch2:
        statement;
        statement;
    ...
    default:
        statement;
        statement;
}
```

safer and more
efficient than
nested if-else
statements

Programs with deeply nested `if-else` statements tend to be difficult to manage. Whilst the syntax may hold, mistakes in indentation may cause the programmer to interpret the logic incorrectly. Nested `if` statements are also inefficient, since they require many expressions to be tested in order to match conditions at the end of the sequence.

C++ provides a statement specifically for multi-way decision making. The `switch` statement allows a number of branches to be defined, and jumps directly to the branch that matches the given condition.

The keywords associated with the `switch` statement are `case`, which defines a branch condition, and `default` which defines a wild-card or catch-all condition.

In the above `expr` must be an integral expression. Once evaluated control passes to the first matching `case` expression. All statements are then executed until the end of the `switch` statement.

Note that the `default` condition does not have to exist. Control is transferred here if none of the other `case` conditions match. If it is absent, control is passed to the next statement after the `switch`.

switch Statement

```
#include <iostream>

int main()
{
    int i = 0;
    while (1)
    {
        std::cout << std::endl << "> ";
        std::cin >> i;
        switch (i)
        {
            case 1:      std::cout << "on ";
            case 2:      std::cout << "off ";
            default:     std::cout << "bang ";
        }
    }
}
```

> 1
on off bang
> 2
off bang
> 3
bang

Use the break statement to jump out

In response to different input codes, the switch statement matches on one of the case conditions. Note that when case 1 is matched, all of the statements to the end of the switch statement are executed.

Switch details

The switch expression must evaluate to an integral value

The case conditions must be constants

Each case condition must be unique

default is optional, it is executed if the other statements fail

After the code for one case is complete, execution falls through to the statements of the next case condition (this allows multiple case conditions to be associated with the same set of statements).

Use the break command to stop execution falling through to the next condition. break causes control to jump out of the switch statement.

switch Statement

```
// days should be entered between 0 and 6

std::cout << "$ ";
std::cin >> day;
switch (day)
{
    case 0:      std::cout << "Monday" << std::endl;
                  break;
    case 1:      std::cout << "Tuesday" << std::endl;
                  break;
    case 2:      std::cout << "Wednesday" << std::endl;
                  break;
    case 3:      std::cout << "Thursday" << std::endl;
                  break;
    case 4:      std::cout << "Friday" << std::endl;
                  break;
    case 5:
    case 6:      std::cout << "Weekend" << std::endl;
                  break;
}
```

\$ 0
Monday
\$ 1
Tuesday
\$ 5
Weekend
\$ 20
\$

In this code fragment, weekday codes from 0 to 6 are translated into their corresponding names. "Weekend" is used to represent Saturday and Sunday codes.

Course Notes



Chapter 6: Looping Constructs

- **while Statement**
- **do Statement**
- **for Statement**
- **break, continue & goto Statements**

while Statement

- The while statement is an entry-condition loop

```
while (conditional expression)
    statement;
```

- Condition is evaluated
 - before entering the loop
 - before each subsequent iteration
 - loop iterates while condition is true
- The loop body may be a compound statement

The `while` loop iterates around its body statement until the conditional expression is false. `while` is an entry condition loop. This means that the expression is tested before the loop is entered. If it is false then control is immediately passed to the next statement after the loop, without a single iteration.

The statement indicated in the above syntax may be a compound block. Just as with previous constructs (such as `if`), braces should be placed around the group of statements to which the construct applies.

while Statement

- Condition is true if non-zero

```
int i = 10;
while (i > 0)
    std::cout << i--;
```

```
int i = 10;
while (i)
    std::cout << i--;
```

- Comma operator may be used

```
// code fragment
while (std::cin >> ch, ch != '.') {
    switch (ch)
    {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u': ++vowel;
    }
}
std::cout << "You entered " << vowel << " vowels" << std::endl;
```

In all of the above loops only one statement is present in the body. Although the `switch` statement is made up of other statements, it still constitutes one statement.

The first examples show the use of `i` as the loop control. The examples are identical except that the second uses the knowledge that 0 is false to save the extra evaluation with the relational operator.

The next example shows the use of the comma operator. Recall that expressions involving the comma operator evaluate to the result of the last term. In this example characters are read into the program until the end-of-line is reached. For each character a test is applied to determine if it is a vowel, in which case the `vowel` variable is incremented.

Note the use of multiple `case` statements, and the absence of a default case (since we're only concerned with vowels). Also note that the characters are in quotes (otherwise they would be considered variables).

while Statement

- Use compound blocks for multiple statements

```
a=0, b=0;

while ((a < 1 || a > 10) || (b > 10 || b < 1))
{
    std::cout << "enter two numbers between 1 and 10: ";
    std::cin >> a >> b;
}
```

- Spot the error

```
while (a > 10);
{
    std::cout << a;
    func();
}
```

The compound block enables multiple statements to appear in the body of the loop. The condition for the loop is made up of several logical and relational operators. In this example, the brackets are in place to aid readability.

There are two errors in the second example; both would cause the loop to run forever. The first error is the semi-colon on the same line as the `while` statement. This is an empty statement and is taken as the loop body. Since the value of `a` cannot change, the loop iterates forever. Even if the semi-colon was removed, however, the second error would still keep the loop iterating forever. The loop control variable has been ignored and is never decremented.

do while Statement

- The do-while loop is an exit-condition loop

```
do  
    statement;  
while (conditional expression);
```

- **Condition is evaluated**
 - after the loop is traversed
 - after each subsequent iteration
 - loop iterates while condition is true

The **do-while** is similar to the **while** statement, except that it is an exit condition loop. This means that the condition is evaluated after each iteration of the loop, not before. The consequence of this is that the loop body is guaranteed to be executed at least once.

while Versus do-while

```
a=0, b=0;

while ((a < 1 || a > 10) || (b > 10 || b < 1))
{
    std::cout << "enter two numbers between 1 and 10: ";
    std::cin >> a >> b;
}
```

- **do-while is particularly useful when the loop control is setup inside the loop body**

```
// a=0, b=0;

do {
    std::cout << "enter two numbers between 1 and 10: ";
    std::cin >> a >> b;
} while ((a < 1 || a > 10) || (b > 10 || b < 1));
```

Although the `while` loop is used more commonly than the `do-while`, the `do` loop is very useful in certain situations. It is used when the loop control needs to be setup within the loop body.

In the `while` loop above `a` and `b` are setup with dummy values to ensure that the loop is iterated once and the variables are setup properly (this is better than the alternative of replicating the `cout` and `cin` statements before reaching the loop). This is unnecessary in the case of the `do-while`, however, since one iteration is guaranteed.

for Statement

- The **for** loop is an entry-condition loop
 - formalises loop components of setup, test and update

```
setup  
while (test)  
    update;
```

```
int i = 0;  
while (i < 10)  
    ++i;
```

- **for** is the most powerful and commonly used loop

```
for (setup; test; update)  
    statement;
```

```
for (int i=0; i<10; ++i)  
    ;
```

The **for** loop is an entry condition loop which formalises the initialisation, test and update of the loop control. Three expressions are defined in the **for** statement for this purpose.

The setup expression is executed once when the **for** statement is reached. The test is executed with each iteration of the loop and before the first. Thus the loop is only entered if its condition is true. The update expression is executed at the end of each iteration. It is often used to adjust the loop control variable so that the loop eventually terminates.

The expressions in the setup, test and update parts of the **for** loop may be quite complex. It is not uncommon to find **for** loops with only an empty statement as a body.

for Statement

```
#include <iostream>

int main()
{
    for (int i=0; i<10; ++i)
        std::cout << i << " ";
}
```

0 1 2 3 4 5 6 7 8 9

```
#include <iostream>

int main()
{
    char name[] = "C++ Programming";
    for (int i=0; i<sizeof(name); std::cout << name[i++]);
}
```

C++ Programming

for loops are often used with arrays and pointer types to index data structures

In the second example above the loop moves the variable `i` through the range of valid indices for `name`. In the update expression it is used to subscript the array and output its contents.

for Statement

```
for (int i=0, j=0; ...  
for (start = 10; ...  
for (char *p1 = getName(); ...  
for ( ; ...
```

initialisation, may
be a declaration or
expression

```
for (...; index < arraySize; ...  
for (...; i; ...  
for (...; *p1++ = *p2++; ...  
for (...;; ...
```

loop control, evaluated
before each iteration

```
for (...; ...; ++i )  
for (...; ...; ++a, ++b, ++c )  
for (...; ...; )
```

update, evaluated after
each iteration

```
for (;;)
```

an infinite loop

The examples involving pointers (p1 and p2) will be clearer later in the course. However, the first initialises a pointer to some address returned by `getName()`. The second example, in the test expression, copies whatever p2 is referencing into p1, and increments both pointers.

Notice that any of the three expressions used by the `for` loop may be absent. In the last example all are missing, and consequently this is an infinite loop.

Jump Statements

- **Jump statements unconditionally transfer control**
- **break statement**
 - terminates current while, do, for or switch statement
- **continue statement**
 - terminates iteration of current while, do or for loop
- **goto statement**
 - transfers control to a target label within the current function
 - rarely used, and considered poor programming practice!

Jump statements cause the transfer of control from the point of the jump to some other place. `break` and `continue` are generally considered respectable, and are often used in loops and `switch` statements. `goto`, perhaps the most infamous of all statements in all languages, should only be used as a last resort -- perhaps to jump out of deeply nested loops in the event of an error. Even this use is somewhat redundant in C++, since the formal exception handling facility provides a more robust mechanism.

break & continue

- Used to prematurely continue or exit a loop

```
for (;;)
{
    char input;

    std::cout << "Starting ..." << std::endl;
    std::cout << "$ ";
    std::cin >> input;
    if (input == 'c')
        continue;
    else if (input == 'b')
        break;
    std::cout << "looping ..." << std::endl;
}
std::cout << "finished" << std::endl;
```

```
Starting ...
$ c
Starting ...
$ a
looping ...
Starting ...
$ b
finished
```

In the example, `continue` causes the loop to transfer control to the start; and `break` causes the loop to be terminated.



Chapter 7: Functions & Program Structure

- **Structuring Programs**
 - **Using Functions**
 - **Function Declarations**
 - **Passing Arguments into Functions**
 - **Pass by Value, Pointer & Reference**
 - **Returning Results from Functions**
 - **Function Overloading**
 - **Function Inlining**
 - **Scope**
 - **Program Organisation**
 - **C++ Linkage**
-

Structuring Programs

- **C++ programs contain a number of functions**
 - `main()` is where program execution begins
 - user-defined functions modularise the program
 - library functions provide 'off-the-shelf' utilities
 - class libraries encapsulate data and functions to form objects
- **C++ functions**
 - must be declared before use
 - cannot be nested within one another
 - may be defined in separate files

Most high-level languages structure a program into independent parts called functions or procedures.

C++ provides facilities for general functions which can take and return arguments. The primary purpose of functions in C++ is to provide behaviour for user-defined data structures in classes.

Using Functions

declaration



definition



call



```
#include <iostream>

int getnum()
{
    int number;
    std::cout << "Please enter a number: ";
    std::cin >> number;
    return number;
}

int main()
{
    int a, b;
    a = getnum();
    b = getnum();

    if (a > b)
        std::cout << a << " is the largest";
}
```

Functions may not be nested but must be defined or declared before use. In the above, the definition (the body of the function) and its declaration (the name and types of the function) appear before it is called.

There are no parameters passed to the function. It merely prompts the user for input and returns this to the calling function using the return statement. `return` causes the function to terminate immediately and evaluates to the returned value. Thus functions returning values may appear in expressions.

Forward Declarations

declaration →

call →

definition →

```
#include <iostream>

int getnum();

int main()
{
    int a, b;
    a = getnum();
    b = getnum();

    if (a > b)
        std::cout << a << " is the largest";
}

int getnum()
{
    int number;
    std::cout << "Please enter a number: ";
    std::cin >> number;
    return number;
}
```

on the side

Function and variable declarations tell the compiler about names and types.

Function and variable definitions fully describe functions or allocate space for variables.

It is usual in C++ for the functions to appear after `main()`. If they are called before the definition a declaration must be provided. The declaration specifies the functions name, return type, and argument types (if any).

Function Arguments

- Values are passed into functions via argument lists

<pre>int add (int a, int b) { return a + b; } // ... int result = add (4,5);</pre>	<p>formal argument list</p> <p>actual parameters</p>
---	--

- **Formal argument list**
 - identifies the types, order, number of the arguments
 - may also associate a name with the arguments
- **Actual parameter list**
 - must provide the correct types and number of arguments
 - parameters may be literals, variables or expressions

Formal parameter lists, providing details of argument types and numbers, allow the compiler to verify that the function is being called correctly.

Function Arguments

```
#include <iostream>

int max (int, int);

int main()
{
    int num1, num2;
    std::cin >> num1 >> num2;
    std::cout << "The largest is "
               << max (num1, num2);
}

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

← In the declaration, variable names are optional

← Arguments names are local to the function (scoped)
Only position is significant

When providing function declarations it is not necessary to name the arguments, only the types are important. In fact, it is sensible not to name the arguments since the compiler does not check whether these match the names later used in the definition.

In the program the `cout` statement is broken over two lines. Statements may be broken at any convenient point (except in the middle of operators and variable names).

The ternary-`if` is used in the example to provide a succinct mechanism of making the comparison. The usage is quite common.

Function Prototypes

- **Function declarations are ‘function prototypes’**
 - provide return types and argument types

```
void display (int);

int main()
{
    int x = 0;
    display (x);
}

void display (int i)
{
    std::cout << i;
}
```

- **Functions not returning values are of type void**
 - the return statement, if used, should not have a value

Function prototypes are used in both ANSI C and C++. Function prototyping requires that all functions are at least declared (otherwise they must be defined) before use. The types of all arguments and the function return type should be specified.

Note that functions not returning a value are of type void. Functions not receiving arguments should use empty parenthesis.

```
void function ();           // c++ void arguments
void function (void);       // ANSI C void arguments
```

Using `f(void)` to indicate no arguments is anachronistic in C++ but is generally allowed. Using the C++ style `f()` in C means that `f` can take any number of arguments of any type, and is therefore quite wrong.

Compilers are only required to check type consistency between function declarations and definitions. They do not check argument naming, so inconsistency and confusion may exist here. To avoid the problem, do not supply argument names when only declaring functions.

Function Name Overloading

C++ allows multiple functions to share the same name

```
void display ();  
void display (int);  
void display (int, float);  
void display (float, int);  
  
int main ()  
{  
    display();  
    display(1);  
    display(1, 4.3);  
    display(3.4, 3);  
    // display(3, 5);           // ambiguity  
}
```

Improves clarity where the same logical operation is being applied

Avoids pollution of the name space

Function overloading enables multiple functions to share the same name provided they can be distinguished by their parameter lists. The return type of the function is not considered when determining the function to be executed.

Ambiguities can exist in programs when arguments have to be converted to match a function. Where two equally favourable paths exists an ambiguity is raised by the compiler.

The compiler applies a set of matching rules when coercing arguments to match functions.

Default Parameter Values

Default values are used when insufficient arguments are provided.

Default values should be defined from right to left. Actual parameters are supplied from left to right.

Default values are part of a functions interface and therefore appear in the declaration.

```
void setCursor (int, int = 0);

int main ()
{
    // setCursor ();
    setCursor (1);           // x=1, y=0
    setCursor (1,3);        // x=1, y=3
}

void setCursor (int x, int y)
{
    update_cursor_position (x,y);
}
```

C++ allows default parameter values to be supplied when functions are called with insufficient arguments. The default values must be specified with the function declaration (i.e., in a header file not a separately compiled unit), and apply from right to left.

The facility of default values is similar to function overloading, and is especially useful in situations where function overloading would be used to allow different numbers of arguments to be supplied.

Beware of ambiguities introduced by providing default values for functions which are also overloaded.

```
void ping ()
void ping (const char *, int)
void ping (int)
void ping (const char * = "", int = 4)

ping ()           // ambiguous, which ping is called?
```

This last prototype also demonstrates the syntax to use when providing default values in declarations where argument identifiers are not supplied (the norm). Beware that extra spacing must sometimes be used to avoid confusion with operators. For example, the above could have been written

```
ping (const char *= "", int = 4)
```



This would fail to compile since an argument list in a function declaration is an inappropriate place for the *= operator!

Call by Value, Pointer & Reference

- **Call semantics relate to the way a function interacts with the program**
- **Call-by-value**
 - only copies of the parameter values are available to the function
 - only one value can be passed back through the return statement
- **Call-by-pointer**
 - the addresses of parameters are passed to the function
 - parameters can be changed by dereferencing the addresses
- **Call-by-reference**
 - references (aliases) of the parameters are made by the function
 - changes are propagated back directly

The functions presented so far are limited in that they can only return one value to the caller. A more general way of passing information both into and out of a function is to use the formal argument list.

Depending upon the call semantics (and consequently the function definition), arguments can be declared to pass information bi-directional.

There are three ways of passing arguments to functions; by value, by pointer and by reference. By value causes a copy of the contents from the calling parameters to be passed to the function. Changes can therefore only be made to the copy and have no effect on the original variables. By pointer causes an address to be passed to the function. The address must be dereferenced in order to read the data; and since this is the address of the original parameter, to also write data. By reference (or alias) allows the function to make a synonym for the parameter, which can be used as if the parameter was available locally.

Call by Value

- Function takes copies of contents of arguments

```
void swap (int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

int main()
{
    int x = 5, y = 6;
    std::cout << x << " / " << y << std::endl;
    swap (x,y);
    std::cout << x << " / " << y << std::endl;
}
```

5 / 6
5 / 6

Values are not swapped
because function cannot
access the variables x and y

Note that the call simply includes the names of the parameters whose values are passed to swap.

Call by Pointer

- Function takes copies of addresses of arguments

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

int main()
{
    int x = 5, y = 6;
    std::cout << x << " / " << y << std::endl;
    swap (&x, &y);
    std::cout << x << " / " << y << std::endl;
}
```

5 / 6
6 / 5

The values copied into swap are addresses. Once dereferenced they are the same as x and y

Notice that the address-of operator is used to take the addresses of the parameters and pass these to `swap`. Since `swap` receives addresses it must dereference them in order to access the data. This gives `swap` access to the actual variables so the function succeeds.

Call by Reference

- Function makes aliases to arguments

```
void swap (int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}

int main()
{
    int x = 5, y = 6;
    std::cout << x << " / " << y << std::endl;
    swap (x,y);
    std::cout << x << " / " << y << std::endl;
}
```

5 / 6
6 / 5

The pointer mechanism is
implemented internally

Call by reference is similar to variable parameters in Pascal and Modula2. The called function takes a reference to the argument being supplied. Therefore the function is acting upon the real arguments, not copies.

Call by Reference

- **Pros**

- as efficient as using pointers
- less cumbersome than pointers
- no & required by caller or * required by function
- reduces referencing errors

- **Cons**

- harder for C programmers to read
- requires function declaration to distinguish between call by value and call by reference

function (employee)

Call by reference
or call by value?

One place where reference parameters are essential is when overloading operators. If the operator needs to change an operand it is unsightly (a syntactic mess) for the caller to have to pass an address. For example, consider the extraction operator used in the istream.

```
int i;  
std::cin >> i;
```

In this the message extract (>>) with a parameter `i` is sent to the object `cin`. The corresponding function takes the next integer off the input stream and stores it in `i`. This works because operator `>>` takes a reference to the right hand operand. Using C style pointers, the syntax would have become

```
std::cin >> &i
```

Of course, there is a pay back. The `&` operator now has four independent meanings in C++

<code>&p</code>	// take the address of p
<code>a & b</code>	// logical AND
<code>a && b</code>	// relational AND
<code>int&</code>	// an integer reference

Inline Functions

- **Call Overhead**
 - making a function call incurs a small performance cost
 - on small functions this is sometimes unacceptable
 - **Macro Substitution**
 - C++ uses a pre-processor to #include files
 - the pre-processor can also define macros for code substitution
 - macros are dangerous because expressions are not understood
 - **Inline Functions**
 - C++ compilers can also perform code substitution
 - benefit from proper understanding of language
 - use the 'inline' keyword when required
 - note that code size may substantially increase
-

Inline Function Substitution

```
#include <iostream>
#define MIN(a,b) (a < b ? a : b)

int main()
{
    int x=5; y=6;
    std::cout << "min is " << MIN (x,y) << std::endl;
    std::cout << "min is " << MIN (++x, ++y);
}
```

min is 5
min is 7

```
#include <iostream>
inline int MIN(int a, int b) { return a < b ? a : b;}

int main()
{
    int x=5; y=6;
    std::cout << "min is " << MIN (x,y) << std::endl;
    std::cout << "min is " << MIN (++x, ++y);
}
```

min is 5
min is 6

The `inline` keyword is a hint to the compiler to use code substitution in place of the usual function call implementation. This is C++'s replacement for pre-processor `#macros`.

- compilers may choose to ignore the `inline` request when the function is large (since the call overhead is negligible) or when it detects recursion.
- inlining improves performance when frequently accessing small (macro like) functions; however, it increases code size. Once inlined, the function code is available to optimisers that would otherwise not optimise across a function call.
- inlining does not affect the meaning of a function call, and is therefore safer than a macro.
- inlining is also type safe, since both actual argument types and return types must be declared. In addition, all the usual scope rules are obeyed.

In order to `inline`, compilers need source code to effect a substitution. It is therefore not possible to `inline` between separately compiled modules. Inline functions are usually placed in header files.

Note that inline functions have static linkage. They may therefore be defined in header files without the risk of multiple definitions being visible from separate translation units.

Scope

- Refers to the visibility of objects

```
void swap (int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}

int global;

int main()
{
    int x = 5, y = 6;
    std::cout << x << " / " << y << std::endl;
    swap (x,y);
    std::cout << x << " / " << y << std::endl;
}
```

**a, b and temp are
local and only visible
to swap**

**global is visible
throughout the entire
program**

**x and y are only
visible to main()**

All variables must exist in some scope. C++ allows for many scopes. In the example above there is global scope (accessible to all) and function scope, private for `swap()` and for `main()`.

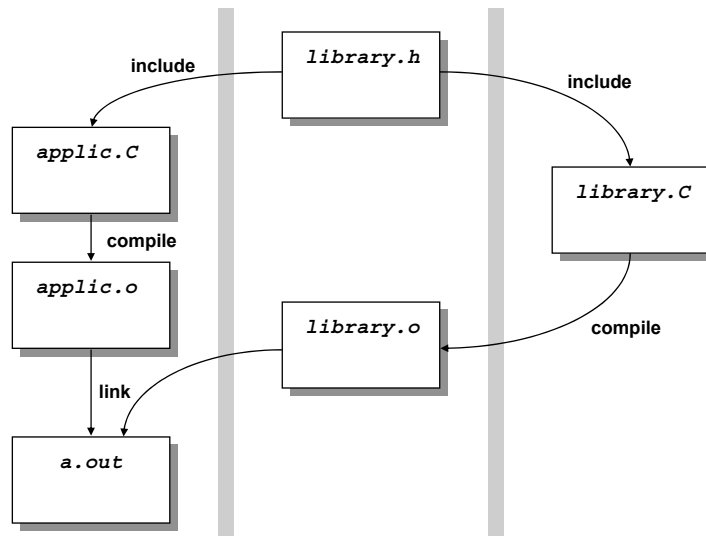
Program Organisation

- **C++ programs are organised as a set of files**
 - library files hold function definitions
 - header files hold function prototypes
 - program file hold the user's program
- **Compile the library and program files**
 - compile to the intermediate object file (.o)
 - link files together to produce the executable

Programs are constructed in several files for pragmatic reasons. They are often very large and may require several people to be concurrently working. Modularising the program among files also helps to maintain high cohesion within related functions, and low-coupling between unrelated functions.

Most programs are organised around libraries, either developed specifically for the project or taken from standard locations. Both C and C++ have comprehensive libraries supporting the languages.

Program Organisation



`applic.C` holds the central part of the program and the function `main()`. `library.C` holds various functions providing tools for the program. `library.h` contains function declarations required to be included into `applic.C` and `library.C` so they may be separately compiled.

Once compilation is complete, the files are linked together to form an executable program.

Program Organisation

- Program spread over three files

main.C

```
#include <max.h>

int main()
{
    ...
    total = max (x, y);
}
```

max.h

```
int max (int, int);
```

max.C

```
#include <max.h>

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

C++ Linkage

- **C++ mangles function names to support overloading**
 - names are augmented with argument types
 - mangled names enable type safe linking
- **To link C code, use C linkage wrapper**
 - this stops name mangling so the linker can find the function

```
extern "C"
{
    int atoi (const char *);
    int printf (const char *, ...);
};
```

```
extern "C" int atoi (const char *);
```

C++ mangles function names to include argument type information. This ensures that the linker matches calls to functions when linking between modules, and also ensure type security.

A consequence, however, is that in order to link to C code the compiler must be told not to mangle the C function names. This is achieved by declaring C functions within a C linkage wrapper. C function names are thereby passed directly to the object file to be resolved by the linker.

In the future, the construct should also provide for ADA and FORTRAN linkage.



Chapter 8: Pointers & Arrays

- Simple Pointers
 - Simple Arrays
 - Pointer Arithmetic
 - Using Pointers as Arrays
 - Using Arrays as Pointers
 - Passing Arrays to Functions
 - Pointer Idioms
 - Pointers & Multi-dimensional Arrays
 - Void Pointers
 - Constant Pointers
 - Promotion to const
 - Function Pointers & Tricky Definitions
-

Simple Pointers

- **Pointers are variables which hold addresses**
 - their type is determined by what they point at

on the side

Recall that the & operator takes an address, and the * operator dereferences an address.

Don't confuse * in the pointer declaration, with the * operator.

```
#include <iostream>

int main()
{
    int i = 5;
    int *p = &i;
    *p = 10;
    std::cout << *p << " " << i << std::endl;

    int **j = &p;
    int ***q = &j;
    int ****r = &q;
}
```

i	10	100
p	100	104
j	104	108
q	108	112
r	112	116

- **How can you print the value of i via r ?**

Pointers are like any other variable. A pointer is a region of memory in which values may be stored. The interesting thing about pointer variables is that they all hold addresses as values. Pointer variables differ from conventional variables, in that their type is determined by the type of the objects which they can point at.

In the example above, an integer variable *i* and an integer pointer *p* are defined in *main*. *i* is a variable typed to hold an integer; *p* is a variable typed to hold the address of an integer variable.

Three other pointers are also introduced into the program. Although they may look complex, the idea is exactly the same as for the simple pointer *p*. Specifically, *j* is a variable able to hold the address of a variable which is able to hold the address of a variable able to hold an integer. *j* is a pointer to a pointer to an integer. *q* and *r* simply introduce more levels of indirection. To print the value of *i* through *r*, apply the dereferencing operator to *r* four times. That is, simply follow the pointer through four levels of indirection; in the above picture, imagine this as four hops from the bottom box to the top.

Simple Arrays

- **An array variable is a contiguous vector**
 - a collection of objects of the same type

```
#include <iostream>

int main()
{
    int array [3];
    array [0] = 10;
    array [2] = 20;

    char matrix [3] [2];
    matrix [0] [0] = 'a';
    matrix [2] [1] = 'z';
    matrix [2] [0] = 'w';
}
```

array[0]	10	100
array[1]		104
array[2]	20	108

	[0]	[1]
matrix[0]	a	
[1]		
[2]	w	z

matrix[0][0]

a				w	z
112	113	114	115	116	117

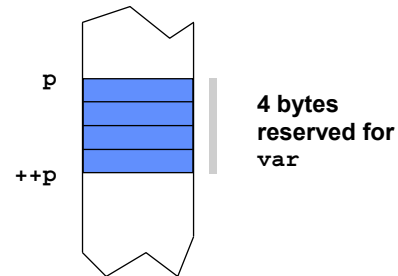
An array variable is a variable able to hold multiple-values of the same type in an ordered, contiguous, sequence. In the example code above, memory is allocated for the three integer elements of array as indicated by the figure. A two dimensional array is also stored as a contiguous sequence, with successive rows from the structure appearing in sequence in memory.

Arrays are accessed by means of the subscript [] operator. The supplied argument determines the offset from the base of the array of the desired element. In the case of multi-dimensional arrays, (for example, `matrix[row][column]`), the offset is determined from both row and column arguments.

Pointer Arithmetic

- Pointers may be manipulated according to their type
 - a unit equals the size of the underlying object type

```
int main()
{
    int var;
    int *p = &var;
    ++p;
}
```



```
int main()
{
    int i=2, j=3;
    int *p = &i;
    std::cout << *(p-1) << std::endl;
}
```

What is the output?

The type of a pointer is determined by what it is able to point at, rather than characteristics of the pointer's memory itself. The reason for this is pointer arithmetic. This powerful notion in C and C++ allows pointers to be manipulated (incrementing and decrementing addresses) and used in relational expressions (comparison).

When a pointer is used in an arithmetic expression, the result depends on the underlying type of the pointer. For example, when an integer pointer is incremented, its value is increased by the size of an integer (usually 4 bytes). When a character pointer is incremented its value changes by 1 byte. The idea is that when you look at memory through a pointer of a certain type, then the size of the underlying type becomes the arithmetic unit.

In the above code fragment three variables *i*, *j* and *p* are declared on the stack frame of *main*. *p* holds the address of *i*, and the stack grows down (negatively) on this system. Due to pointer arithmetic, *p* is displaced by four bytes (1 integer) to refer to *j*. Consequently, 3 is displayed on the output.



The program in the example is actually quite dangerous. In the first instance it relies the stack growing down. In the second, it assumes that *j* follows *i*. If another variable is inserted between the two (perhaps during code maintenance) then the answer will be wrong.

Pointer Arithmetic

- Basic pointer (address) arithmetic & relations

++, --, +, -, =, +=, -=
==, <, >, <=, >= !=, !

```
int main()
{
    char array[] = "hello";
    char *p = array;
    if (&array[2] != (p+2))
        std::cout << "weird!" << std::endl;
    else
        std::cout << "good" << std::endl;
}
```

on the side

Note that the name of an array is evaluated to the address of the first element.

Some of the operators which may be used in pointer arithmetic and pointer expressions are shown above. Note that *, / and % are not allowed. Complex arithmetic could be performed by casting the pointer to an integer, and later casting back.

The example shows two pointers being compared for inequality. First `p` is assigned the address of the first element of array. Note that the name of the array without a subscript resolves to this first address. An equivalent form is

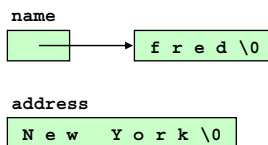
`array == &array[0]`

Next the address of the third element in the array is compared to the address of `p` offset by two integers. These addresses should be identical because an array is allocated from contiguous memory.

Pointers & Arrays

- A pointer is a variable that holds an address
- an array is a vector of variables.

```
int main()
{
    char *name = "fred";
    char address[] = "New York";
}
```



An array name cannot be used as an l-value because it does not have memory

There is often some confusion regarding the difference between pointers and arrays. This is not surprising given that they may often be used interchangeably in C and C++.

Pointers are actual variables which hold the address of some other variable of interest. During their lifetime, pointers may be assigned the addresses of other objects of interest.

Arrays are not pointers. There is not a variable in which the address of the first element of the array is stored. Rather the array is a variable which holds multiple values. Confusion sometimes comes from the fact that the array name may be used without a subscript. As such it evaluates to the address of the first element---the array does not, however, contain this information.

An array cannot be used as an l-value because (without a subscript) it does not evaluate to a location. It evaluates to the address of the first element, and this is not in itself an l-value. If it was treated as a valid l-value, then assignment to an array would cause the first four bytes (in the above example, four elements) to be overwritten.

Pointers & Arrays

- **Arrays may be dereferenced, pointers may be indexed**

```
p[i] == *(p + i)
```

```
#include <iostream>

int main()
{
    char name [] = "hello";
    char *p = name;
    std::cout << p[0] << p[4] << " ";

    std::cout << *name << *(name + 4) << std::endl;
}
```



```
name is not an l-value    →    * (p += 2) = 'L'
                               // * (name += 2) = 'L'
```

Arrays and pointers may often be used interchangeably. In particular, address arithmetic, and dereference and subscript operators may be applied to both. However, contrary to appearance an array is not a pointer and may not be used as a l-value.

In the example above, the address of name (actually, of the first element) is assigned to p. When the subscript operator is applied to p, the index given is added as an offset and then the resulting address is dereferenced.

```
p[0] == *(p+0) == *p
p[2] == *(p+2)
```

Similarly, pointer arithmetic may be applied to the array. Again, this is because the name evaluates to the address of the first element.

```
* (array + 4) == array[4]
```

Passing Arrays to Functions

- **Arrays are passed to functions through pointers**
 - by using the subscript operator the pointers feel like arrays

vec is a pointer and an l-value	→	<pre>#include <iostream> void printVector(char vec[]) { std::cout << vec[0] << std::endl; char temp[2]; vec = temp; } int main() { char vector[2] = {'a', 'b'}; printVector (vector); char temp[2]; // vector = temp; }</pre>
vector is an array, not an l-value	→	

When arrays are passed into functions the call semantics are implicitly call-by-pointer. In the call the array is evaluated to the address of the first element and this is copied to a pointer on the stack of the called function. Since pointers can be used like arrays (via the subscript operator), the illusion is given that the actual array has been copied into the function. In fact, only an address has. A confusion which sometimes exists concerns how to prototype the function. The following two forms for `printVector` are equivalent.

```
void printVector (char *vec);
void printVector (char vec[]);
```

Note that since `vec` is a pointer it may be used as an l-value. In practice such programming is discouraged because it can lead to maintenance problems. A person updating `printVector` may not be aware that this 'input' parameter is corrupted earlier in the function.

Since `vec` is indeed a pointer you cannot determine the length of the original array from within `printVector`. In any case, such size information must be determinable at compile time, but the actual length of the array passed to `printVector` will depend on who makes the call at run-time.

Higher level abstract containers (as defined in the STL) are more commonly used to maintain vectors and lists of objects.

Passing Arrays to Functions

```
#include <iostream>

void printGrid(char matrix[][4])
{
    std::cout << matrix[0][1] << std::endl;
    std::cout << matrix[1][3] << std::endl;
}

int main()
{
    char mat[2][4] = {'a', 'b', 'c', 'd',
                     'e', 'f', 'g', 'h'};
    printGrid (mat);
}
```

b
h

on the side

matrix is actually a pointer on the stack frame of printGrid.

Only col is required to calculate the correct offset in matrix.

```
matrix [1][3] == *(matrix + (1 * 4 + 3))
matrix [r][c] == *(matrix + (r * col + c))
```

Just as with single arrays, two dimensional arrays use pointer semantics when they are passed into functions. In the above example, `mat` evaluates to the address of the first element (first row, first column), and this is passed into the pointer `matrix` in `printGrid`.

The prototype for this kind of pointer needs only to specify the number of columns. Specifying rows is valid, but redundant. The reason that column number is required is so that the compiler can work out the correct offset of the element being indexed. In the above example, `matrix[1][3]` evaluates to the element whose address is

$$\text{matrix} + 1 * 4 + 3$$

Where the 4 is the number of columns in the two dimensional array.

In the above program `matrix` is a pointer variable on the stack frame of `printGrid`. Since pointers are variables which hold addresses, it is possible to assign a suitably typed address to `matrix`.

```
char temp [2][4];
matrix = temp;
```

This is certainly not good programming practice, but it demonstrates that `matrix` is just a variable. Note that the type of the address being assigned to `matrix` is significant. If `temp` was sized `[2][5]`, then the compiler would complain of a type mismatch.

Pointer Idioms

- **Pointers and arrays are often used together**
 - array name defines base of contiguous piece of memory
 - pointer moves through elements

```
void strcpy (char *t, const char *s)
{
    while (*t++ = *s++);
}

int main()
{
    char source[] = "C++ Programming";
    char target[20];
    strcpy (target, source);
}
```

***s++**

- **Which operator is applied first ?**

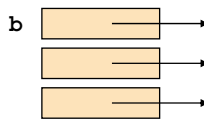
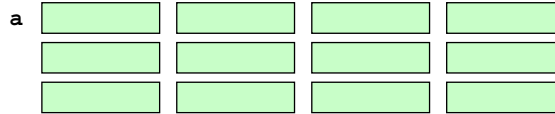
The implementation of `strcpy` is subtle but illustrates a common programming style in C++. Both `s` and `t` are pointers. The body of the `while` is empty. With each iteration of the loop, the value pointed to by `s` is assigned to the location pointed to by `t`, and both `t` and `s` are incremented. While the value of the assignment expression is non-zero, the loop continues. However, since strings in C and C++ are null terminated (`'\0'` or simply `0`), upon copying the null to `*t`, the assignment expression will evaluate to `0` and the loop will terminate.

An important detail concerns how `*s++` is evaluated. The operators have the same precedence and both evaluate from right to left. It follows then that the `++` operator is applied first, and then the `*` operator after. This may suggest that the next element is being dereferenced rather than the current one. However, this is not the case because `++` is applied in a postfix fashion. `s` is incremented to the next element but the expression `s++` evaluates to the old location of `s`. Consequently, `*s++` evaluates to the current element.

Pointers & Multi-dimensional Arrays

- These two declarations differ greatly

```
int a[3][4]
int *b[3]
```



There is no 'int' storage space allocated, only pointers

The advantage of the pointer array is that the rows may be of different lengths

on the side

Note that access to an int from a or b can be identical `a[1][1]`, `b[1][1]`.

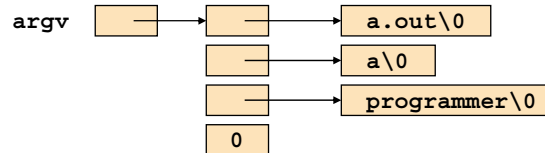
The declarations differ in that the first defines a two dimensional array while the second an array of pointers. In the two dimensional array there are no pointers and all of the necessary memory to accommodate the variable is allocated. In the array of pointers no memory has yet been allocated to hold integers. This has to be claimed for each element (probably dynamically) when necessary.

The advantage of a is that it is ready to use, however, it is somewhat rigid in its structure. This rigidity is overcome in b but at the expense of burdening the programmer. A programmer must allocate memory for each row individually.

The flexibility of b should be immediately apparent. First, none of the rows (single dimensional arrays) need to be of the same size. Second, if it was necessary to reorder the rows in a and b, then in the case of b this could be done by manipulating the pointers. In a the values in the rows would have to be explicitly copied.

Command Line Arguments

- Supplied through an array of pointers (argv)



```
#include <iostream>

int main (int argc, char *argv[])
{
    std::cout << argc << std::endl;
    while (--argc)
        std::cout << argv[argc] << std::endl;
}
```

```
$ ./a.out a programmer
3
programmer
a
```

Command line arguments are passed into C and C++ programs through the argument vector `argv`. This is a pointer to an array of pointers to arrays of characters. The argument count, `argc`, is also passed into a program.

Void Pointers

- Void pointers can point to any type of data

```
int i = 20;
char c = 'F';
double d = 3.1315

int *ip = &i;
char *cp = &c;
double *dp = &d;

// ip = &d;
// dp = &c;

void *vp = &i;
vp = &c;
vp = &d;

// d = *vp;
d = *(double *)vp;
// dp = vp;
dp = (double *)vp;
```

Pointers of incorrect type

void pointer can point to any type of object

void pointer must be cast before being dereferenced

void pointers provide a means of pointing to any type of data. In some respects they switch off the strong typing model used by C++. In order to dereference a void pointer, it must be cast to a real type.

void pointers are sometimes used in C to separate the logic of a data structure (such as a list) from the type of objects stored. In C++ this is largely avoided due to container classes and parametric types.

However, one common use of void* in C++ is to print the address of an object rather than its contents.

```
char message[] = "hello";

std::cout << message ;           // outputs 'hello'
std::cout << (void *) message ;   // outputs
message address
```

In obscure situations in which functions are unable to define the types of their arguments (i.e., printf from <stdio>), the use of void * is generally avoided. Use a variadic argument instead.

```
int printf (const char *, ...)
```

Take care with void*, an incorrect cast may lead to unexpected results. The use of polymorphism (both inclusive and parametric) tend to remove the need for void* in C++ programs.

Constant Data and Constant Pointers

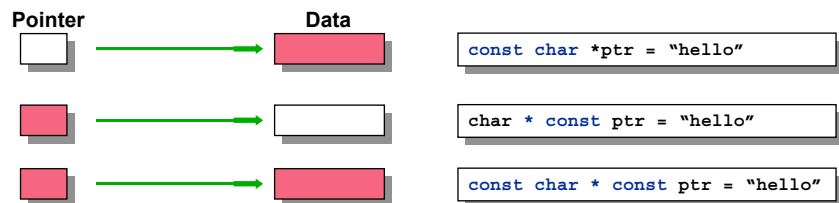
- `const` defines read-only symbolic constants

```
const int SIZE = 20;           // constant data
// SIZE = 21;                 // error
```

- Both data and pointers may be constant

on the side

String literals were of type `char *` in early versions of C and C++. In the current C++ standard they are `const char *`, but for compatibility reasons this type should not be enforced by compilers.



Any data type can be qualified with the keyword `const` to declare objects whose state (once initialised) cannot be changed. A `const` declaration provides a read-only symbolic constant and is often used in place of a pre-processed macro.

Since pointers are variables, they too may be declared to be constant. Thus C++ supports the idea of having pointers to data (`*p`), pointers to constant data (`const *p`), constant pointers to data (`* const p`) and constant pointers to constant data (`const * const p`).

Constant Data and Constant Pointers

```
int main()
{
    char const *p1 = "hello";           // ptr to const data
    // p1[0] = 'A'                       // error
    p1 = "goodbye";

    char * const p2 = "hello";          // const ptr to data
    p2[0] = 'A';
    // p2 = "goodbye";                  // error

    char const * const p3 = p2;         // const ptr to const data
    // p3[0] = 'A';                     // error
    // p3 = "goodbye";                  // error
}
```

Read the declarations from right to left

Beware, qualifiers may be placed in a different order

Constant objects (or variables) must be initialised since (by definition) once they exist their values cannot be changed.

Note that the const qualifier may appear either side of the type, and in even more obscure places by some compilers.

```
const char *p
char const *p
```

The forms are considered exactly equivalent.

Promotion to const

```
int main()
{
    int x = 5;
    const int *xp = &x; // promotion to const
    ++x;
    // ++(*xp);        // const, cannot change

    const int y = 5;
    // int *yp = &y;    // demotion from const
    // ++y;             // const
    const int *yp = &y;

    // const int z;     // must be initialised
}
```

- **Beware forcing const demotion with conversion**
 - non volatile variable may be in a register or read-only memory

```
const float f = 5.5;
const float *fp = &f;
*(const_cast<float*>(fp)) = 10.5;
std::cout << f << std::endl;
```

C++ allows objects to be promoted. It is therefore possible to address a non-constant object through a const pointer. This is safe because the programmer can only do less through the const pointer than through the original reference.

The converse, however, of allowing const data to be accessed through non-const pointers is not possible because it constitutes demotion. If this was possible, an object which was previously declared const and therefore read-only, could be accessed and possibly changed through the new pointer.

Although C++ encourages the use of const, and the compiler enforces the rules of constant data, it is possible to cast const away. However, since this requires explicit action on the part of the programmer, it is unlikely that this would be done accidentally.

It is possible to demote the const data through a pointer to non-const, ideally using the const_cast<> casting operator. The concern here is that the const data may have been placed in a register, as it was defined as const. Generally this style of programming should be avoided, or is very low level. However, if this form of demotion cannot be avoided, then make the variable volatile. Note that this is unnecessary for objects of classes with a constructor or destructor, as such compiler optimisation is not permitted.

Passing Constant Parameters

```
#include <iostream>
#include <iomanip>
#include <stdlib.h>

int strcmp (const char *, const char *);

int main()
{
    char password[100];
    std::cin >> std::setw (100) >> password;
    if (strcmp (password, "hasta la vista"))
    {
        std::cerr << "bad password" << std::endl;
        exit(1);
    }
}
```

const parameters prohibit the function from changing the arguments

const parameters make the function more accessible, non-const arguments are promoted

Far from constraining, **const** declarations tend to make functions more accessible to the underlying program. The function prototype for `strcmp` above enables the function caller to supply a constant argument. If it had been declared without the use of **const** then only non-constant arguments would be permitted.

In order to make functions (and especially member functions of classes) as useful in as many circumstances as possible, it is wise to define all arguments whose values remain unchanged as **const**.

Passing Constant Parameters

- **References add a further complication**
 - what is referenced if the actual parameter type does not match?

```
#include <iostream>

void calc (const int& i, const int& j)
{
    std::cout << i*j << std::endl;
}

int main()
{
    float f=3;
    int j=2;
    calc (f,j);
}
```

- If a conversion exists, then an anonymous variable is created
- but only a constant reference may be taken to this.

References work in a similar way to pointers, they are (after all) simply automatic pointers. Since most C++ programs deal with complex (class) types, a function will commonly take a `const&` to the supplied argument. This makes the function more accessible and safe (assuming it doesn't have to change the argument).

The complication introduced by references occurs when an actual parameter's type differs from the formal argument type. If there is no conversion between the types then compilation will fail. If there is a valid conversion (as above) then the compiler will create an anonymous variable to hold the converted value. The reference then refers to the anonymous variable. Since an anonymous cannot be used as an l-value, the compiler will complain if the reference is not `const`.

Function Pointers

- **Pointers that hold the address of a function**
 - allow the function to be invoked indirectly through the pointer
 - remove the decision of which function to call from call site

```
int foo();
int bar(int (*)());

int main()
{
    int (*p)() = foo;
    foo();
    p();
    bar(foo);
}

int bar (int (*f)())
{
    f();
}
```

bar takes `int (*)()` parameter

`p` is just like any variable, initialised with address of `foo`

function call operators `()`

bar invokes function indirectly

reinterpret_cast

- **Follow normal type rules ...**
 - for argument promotion and conversion
- **To cast between incompatible pointer types**
 - use the `reinterpret_cast`

```
void foo();  
  
void bar()  
{  
    int (*f)();  
  
    // f = foo;  
    f = reinterpret_cast<int (*)>(foo);  
}
```

Useful for low-level programming

Produces implementation dependent code

Tricky Pointer Definitions

```
#include <iostream>

void foo ();

void (*bar (void (*))) ();

void foo () {
    std::cout << "foo" << std::endl;
}

void (*bar (void (*f)())) () {
    return f;
}

int main () {
    void (*fp) () = foo;
    void (*(*bp) (void (*)())) () = bar;
    bp (fp) ();
}
```

use typedef to help ...

```
typedef void (*FP) ();
typedef FP (*BP) (FP);
FP fp = foo;
BP bp = bar;
bp (fp) ();
```

```
$ ./a.out
foo
```

```
void (*(*bp) (void (*)())) () = bar
```

This is a little tricky but is easily resolved when the iterative aspect of the definition is clear. In any case, typedef can be used to simplify.

Note the double function call at `bp (fp) ()`, the first returning the address of the function which is then called (using the function call operator) to generate the output `foo`.

Course Notes



Chapter 9: Object Storage

- **Storage Classes**
 - **Automatic Objects**
 - **Static Objects**
 - **Using Static Objects**
 - **Dynamic Objects**
 - **Free-Store Operators**
 - **Using Dynamic Objects**
 - **Object Lifetimes**
-

Storage Classes

- **Storage class relates to how data are stored**
 - static objects
 - automatic objects
 - dynamic objects
- **All variables (or objects) belong to a storage class**
 - depends on where object is declared
 - depends on qualifications given during declaration
- **Register variables**
 - useful when a variable is heavily used
 - hint to the compiler to assign a physical register
 - not often supported or acted upon by compilers

Storage class relates to where data is actually stored. There are three possible places in a C++ program, in the data segment, on the stack or in the free store heap.

A register declaration is an auto declaration, together with a hint to the compiler that the variables declared will be heavily used. The compiler may ignore the hint, or where possible will associate a physical register in the processor to accommodate it.

```
register int i;
```

The qualifier may only be applied to automatic variables.

Automatic Objects

- **Automatic variables are stack variables**
 - created when entering a function, destroyed on leaving
 - exist on the stack in a 'stack frame'
 - function parameters are also automatics

**i and local are autos
on func's stack frame**



**auto is an automatic
on main's stack frame**



```
#include <iostream>

void func (int i)
{
    int local;
}

int main ()
{
    int auto_on_stack;
}
```

Automatic variables are so named because they live in the context of a function; they come into existence when the function is called and are destroyed when it exits. Autos are physically stored on the stack in a data structure known as the stack frame. The structure holds various information for the function, including the address to which it should return when exiting.

Formal arguments are also automatics and live on the stack frame associated with the function.

Note that since automatics are destroyed when a function terminates, it is not possible to use them to store information which must be preserved beyond the actual function.

Static Objects

- Static objects are established by the compiler and exist in the data segment of the program.

implicitly static



static, but in the scope of main()



```
#include <iostream>

int global;

int main ()
{
    int auto_on_stack;
    static int data_seg;
}
```

- determined by the compiler
- variables exist for the duration of the program
- variables are global or declared with 'static' qualifier
- static data is initialised by the compiler to 0

Static objects are so named because their storage is fixed throughout the execution of the program. In fact, the compiler allocates memory (and disk space) for static variables in the program's data segment.

Since the compiler is involved in the allocation of space for static's, it is able to ensure that all memory is initialised. Static variables, unless explicitly initialised otherwise, are set to 0. This default initialisation is not performed for the other storage classes because of the run-time overhead this would introduce.

Static variables are introduced into a program by either declaring the variables outside of any function (i.e., by making them global), or by qualifying the declarations with the keyword static.

Using Static Objects

```
#include <iostream>

int counter ();

int main()
{
    std::cout << "1st call: " << counter() << std::endl;
    std::cout << "2nd call: " << counter() << std::endl;
}

int counter()
{
    static int calls = 0;
    return ++calls;
}
```

```
$ ./a.out
1st call: 1
2nd call: 2
```

**static storage is preserved
between function calls, but is still
within the scope of the function**

Static objects declared within functions persist between function calls. This is because they reside in the program's data segment and not on the relatively volatile stack. Static functions declared within functions are visible only to that function, whereas global variables are visible to the entire program. In the interests of minimising program coupling, and name space pollution, global variables should be avoided.

In the program above a static variable `calls` maintains a count of the number of times the function `counter` is called.

Using Static Objects

- A function to convert from month number to name

```
const std::string& getMonth (int mon)
{
    static std::string name[] = {
        "curious month",
        "Jan", "feb", "mar", "apr",
        "may", "jun", "jul", "aug",
        "sep", "oct", "nov", "dec"
    };

    return (mon < 1 || mon > 12) ? name[0] : name[mon];
}
```

- Note the return type
 - Reference to static is safe (as not on local stack)
 - Const inhibits caller modifying data

The above function using a static array in the scope of `getMonth` to convert month numbers to month names. The array of pointers exists in the data segment of the program and is automatically initialised to the addresses of the string literals.

Using Static Objects

- A function to dispatch a function from a static table
 - useful for building finite state machines

```
#include <iostream>

void zero () { std::cout << "zero" << std::endl; }
void one () { std::cout << "one" << std::endl; }
void two () { std::cout << "two" << std::endl; }
void three () { std::cout << "three" << std::endl; }
void four () { std::cout << "four" << std::endl; }

void foo(int i) {
    static void (*lookup[])() = {zero, one, two, three, four};
    lookup[i]();
}

int main()
{
    foo (1);
    foo (3);
}
```

```
$ ./a.out
one
three
```

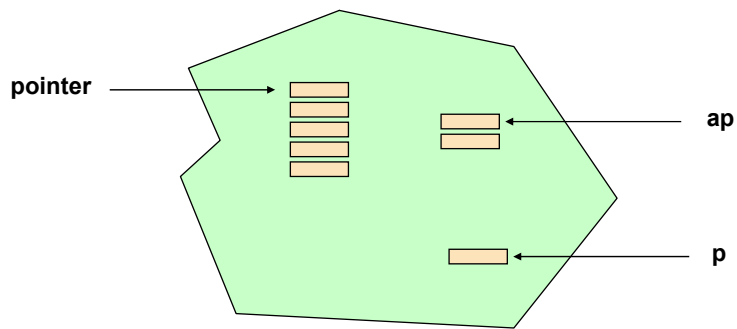
- Initialisation of table delayed until first call

This example program illustrates a common use of function pointers. In this program a dispatch table is setup within foo. The table is setup once (upon first call to foo) since the lookup table is static. lookup is an automatically sized array of pointers to functions that don't take any arguments and return void. lookup is initialised with the addresses of five functions. When called from main, the integer value supplied is used to index the lookup table to determine the appropriate function to call.

The alternative to this is to use a switch statement. The problem with switch is that it makes the code clumsy and less maintainable. Moreover, switch is much less efficient than the lookup table. No matter how many entries are in the table, the array lookup cost remains about the same. However, the cost of traversing a switch may grow linearly with the number of case statements to parse.

Dynamic Memory

- **Memory claimed and released on demand**
 - maintained in a 'free-store heap'



```
int *p = new int;  
int *pointer = new int [5];
```

The free-store heap is an area of memory which programs can dynamically 'dip' into on demand at run-time. Variables are explicitly allocated from the pool of memory, and explicitly returned to the pool.

Dynamic allocation of variables is useful because it allows the program to be more responsive to the memory demands apparent while a program is running, rather than having to predict memory utilisation in advance.

To allocate memory from the heap, C++ provides the `new` operator. Unlike C which provides the standard function `malloc`, C++'s `new` operator is part of the language.

Dynamic Objects

- **Dynamic objects are created on the free store heap**
 - heap space is memory which may be randomly allocated and released by the programmer
 - dynamic objects must be explicitly allocated and explicitly released, and are not related to the stack or data segment
 - dynamic objects do not have identifiers; they simply have addresses which are stored in pointer variables

```
#include <iostream>

int main ()
{
    int *pointer;
    pointer = new int;
    *pointer = 3;
    delete pointer;
}
```



Dynamic objects may be created and destroyed at any time by the programmer. The objects exist in a portion of memory known as the free store heap (or simply the heap).

Objects are allocated from the heap with the new operator and returned to the heap with the delete operator. Since new only returns the address of the object, the object does not have a conventional name. Dynamic objects are referred to through the pointers which hold their addresses.

In the example above, data is allocated from the heap and its address is placed in the variable pointer. Note that the pointer variable itself is an automatic on the stack frame of main().

Dynamic Memory Operators

- Controlling the heap with new and delete

```
int main()
{
    char *ptr1, *ptr2, *aptr;

    ptr1 = new char;           // allocate a single char
    ptr2 = new char ('c');     // allocate and initialise char
    aptr = new char [300];     // allocate an array of char

    delete ptr1, ptr2;         // delete a single item
    delete [] aptr;           // delete an array of items
}
```

dynamic arrays are not initialised

delete 0 is legal

always use [] when deleting an array

new allocates and initialises memory, delete de-initialises and deallocates memory. Initialisation and de-initialisation is important when creating and deleting new objects.

As operators, both new and delete may be overloaded for each class. It is therefore possible to use different allocation schemes depending on the objects being created.

Deallocating Arrays

C++ inherited from C the notion that a pointer points to an individual object, but that object may actually be the initial element of an array. In general the compiler cannot tell. To ensure that the array is correctly deleted (which in C++ means deallocation of memory and a call to the destructor for each element), programmers must explicitly call the delete [] (delete array) operator.

Using Dynamic Memory

- Supporting variable length records
 - with static arrays the size is determined at compile time

on the side

iomanip.h includes definitions of parameterised manipulators, such as setw()

```
#include <iostream>
#include <iomanip>
#include <cstring>

char *getName ()
{
    char name [100];
    std::cin >> std::setw(100) >> name;
    char *p = new char [std::strlen(name) + 1];
    std::strcpy (p, name);
    return p;
}

int main()
{
    char *n = getName();
    std::cout << n << std::endl;
    delete [] n;
}
```

← temporary array on stack

← points to name

← return memory to free-store heap

In this example the function `getName` is used to read in a name (to a maximum of 100 characters), allocate space for the name on the heap, copy the name into this new space, and then return the address of the memory to the caller.

The program demonstrates how dynamic memory allows the memory requirements of a program to change during run-time. By contrast, the use of static or automatic arrays, requires the programmer to anticipate *a priori* the size of various program structures in advance.

As a general style of programming this excerpt is poor in that memory allocated in one part of the program is released elsewhere. In general, it is better to use a class to encapsulate both the allocation and deallocation of such things, and thereby reduce the chance of losing the address, or deallocating the memory multiple times.



Object Lifetimes

	<i>storage class</i>	<i>scope</i>	<i>memory</i>	<i>object lifetime</i>
<code>int global;</code>	static	global	data seg	program
<code>static int file;</code>	static	file	data seg	program
<code>int func (int arg)</code> <code>{</code>	auto	function	stack	function
<code>static int stat;</code>	static	function	data seg	program
<code>int local;</code>	auto	function	stack	function
<code>int *p = new int;</code>	dynamic	pointer	heap	delete
<code>{</code> <code>int block;</code> <code>}</code>	auto	block	stack	block
<code>}</code>				

There are essentially three storage classes (although technically dynamic is often not considered as a true class). A major distinction is the lifetime of objects from different classes:

static	exists in data segment
auto	automatically allocated on stack frame
dynamic	claimed from heap

Scope refers to the visibility of the object

global	program wide (including other files)
file	global within the current file only
function	local to the function block
block	local to the current block

Memory refers to the physical location of the object

data seg	in the data segment of the program
stack	on the stack frame of a function
heap	allocated from the free store heap

Object lifetime relates to when the object is created and destroyed

program	lives while the program is running
function	lives while the stack-frame exists
delete	lives until explicitly deleted
block	lives during the enclosing block

on the side

Objects in a block are created and destroyed as the block is entered and then exited.

Chapter 10: class & struct

- **struct & class Definitions**
 - **Making Objects**
 - **Information Hiding**
 - **Access Regions**
 - **Member Function Declaration**
 - **Member Function Definition**
 - **Organisation of C++ Programs**
 - **inline Functions**
 - **Object Memory Organisation**
 - **The 'this' Pointer**
 - **friend Functions**
 - **Class Scoping Rules**
-

Scalar & Aggregate Data Types

- **Scalar data types contain one item of data**
 - int
 - char
 - long
- **Aggregate data types allow data to be collected**
 - keep together related data
 - array keeps together data of the same type
 - struct keeps together data of different types
- **Abstract data types**
 - build when operations and data are aggregated
 - struct's are classes
 - variables are objects

Aggregate data types enable arbitrary collections of data items to be held together. They are useful for holding related bits of data together and occur in many high-level programming languages. For example, C structs and Pascal records.

In C++ the notion is taken further by allowing the inclusion of functions as members of structures in addition to data. Such structures are termed classes, and are the fundamental building blocks for object-oriented programming.

Struct Definition

- **A struct is a definition of a new data type**
 - in C++, commonly known as a 'class'

```
struct Money
{
    int pounds;
    int pence;
    void display() {
        std::cout << "£" << pounds << "." << pence << std::endl;
    }
};
```

- **encapsulates both functions and data**
- **fundamental in object-oriented programming**
- **models entities in the problem (real) world**

The keyword `struct` or `class` is used to declare or define an abstract data type in C++. The data types are similar to structures in C, with the major exception that C++ data types may contain functions. This ability is a fundamental idea in object oriented programming. Real-world objects can be modelled as they are, collections of both state and operations.

C++ programming is concerned with defining the classes and object interactions which model the application domain.

Making Objects

```
#include <iostream>

struct Money
{
    int pounds;
    int pence;
    void display() {
        std::cout << "£" << pounds << "." << pence << std::endl;
    }
};

int main()
{
    Money mine;

    mine.pounds = 5;
    mine.pence = 90;
    mine.display();
}
```



£5.90

Access members using
the '.' operator



Objects (or variables) of user-defined types can be made and used just as built-in types. Access to the members is obtained through the member selector operators '.' (or -> for pointers to objects).

Note that both data members and member functions are accessed in the same way. Take care to ensure that () are used when identifying a member function, otherwise the address of the function is taken.

Objects of user defined types can be created in static, dynamic or automatic memory, can be passed into functions and returned from functions just like intrinsic types.

```
Money more[100]           // an array of money
sizeof (Money)             // size of money objects
Money Increase (Money)     // Function to raise Money
```

When reading a C++ program, the application of a member function to an object is 'sending a message'. Therefore the following

```
more.display()
```

should be read as sending the message `display()` to the object `more`. It is then up to the object how the message is handled.

Accessing Dynamic Objects

on the side

The pointer member selector operator is often called the 'crows-foot' operator.

```
struct Money
{
    int pounds;
    int pence;
    void display() {
        std::cout << "£" << pounds << "." << pence << std::endl;
    }
};

int main()
{
    Money mine;
    mine.pounds = 5;

    Money *yours = new Money;
    (*yours).pounds = 10;
    yours->pence = 90;
    yours->display();
}
```

Accessing members through the
pointer member selector operator

Access to dynamic objects is gained through pointers to the objects. In the case of dynamic `struct` or `class` objects, individual members can be accessed using the member selector operators. In place of the `'.'` operator (which is rather cumbersome when used with pointers), the crows foot `'->'` operator is usually adopted. This both dereferences the pointer and selects the requested member.

Building an Interface

```
struct Money
{
    int pounds;
    int pence;
    void display() {
        std::cout << "£" << pounds << "." << pence << std::endl;
    }
    void setPounds(int p) { pounds = p; }
    int getPounds() { return pounds; }
};

int main()
{
    Money mine;

    mine.setPounds(10);
    std::cout << mine.getPounds() << std::endl;

    mine.pounds = 50;           // direct access to state data
    std::cout << mine.pounds;
}
```

The interface allows direct access to the data members to be avoided

Classes encapsulate state and behaviour for a particular kind of object. However, when using the object only part of the state and functions are meaningful to the client program. Other parts concern implementation and should not be of interest to the client.

Classes usually define a set of interface member functions (in general object oriented terms called methods) which provide controlled access to the object for clients. The sum of the methods provided by a class implicitly defines the abstraction which the class is modelling.

In the above example, although access methods have been defined, there is nothing to prohibit client code gaining access to the other members.

Abstract Data Types

- **Encapsulates Data and Functions**
- **Defines a type of object from the problems domain**
- **Hides implementation**
 - state changes are achieved through public functions
 - internal workings and state are hidden from user
 - implementation can be changed without affecting user
 - minimal coupling
 - high cohesion
- **Program consists of interacting objects**

`structs` and `classes` are the building blocks of object-oriented systems. They enable things from the problem world to be modelled in software. The `class` (or `struct`) is the blueprint from which objects (instances or variables) are made.

When designing object-oriented systems, it is important to find objects which are meaningful, and define a complete abstraction. They should exhibit a coherent identity. That is, they should be defined to do one thing well, have a minimal interface (to reduce coupling), hide their implementation detail, be complete and work consistently with the other objects in the software model.

Information Hiding

- Access regions hide members in classes and structs

private:	members are hidden from client code
public:	members are visible to client code

```
struct Money
{
private:
    int pounds, pence;
public:
    void setPounds(int p) { pounds = p; }
};

int main()
{
    Money mine;

    mine.setPounds(10);
    // mine.pounds = 50;           // error
    // std::cout << mine.pounds; // error
}
```

Access regions implement information hiding. Members which need to be visible to client code are made public (this is the class interface); members which should not be accessible to client code are made private.

Private members are data and functions which hold the object's state and implement the object's behaviour.



In general, the public interface should be kept as small as possible since this reduces coupling. Each class should be defined to do one thing well, should be a model of one thing in the system. Classes with large interfaces are usually badly designed, and should often be divided into the component things which they represent.

Information Hiding

```
struct <typename>
{
    public member list
};
```

**In a struct all members are
public by default**

```
struct Money
{
    void setPounds(int);
private:
    int pounds;
};
```

```
class <typename>
{
    private member list
};
```

**In a class all members are
private by default**

```
class Money
{
    int pounds;
public:
    void setPounds(int);
};
```

Either `class` or `struct` may be used when defining new data types. In general, however, `class` is used to define data types which have member functions in addition to data members, and `struct` is used when data types only have data members (i.e., the conventional C data type). This convention is recommended by several C++ coding standards specifications.

Function Declarations & Definitions

```
#include <iostream>

class Time
{
    int min, hour, sec;
public:
    void setTime (int, int, int);
    void display ();
};

void Time::setTime (int m, int h, int s)
{
    min = m; hour = h; sec = s;
}

void Time::display()
{
    std::cout << hour << ":"
               << min << ":";
    << sec << std::endl;
}
```

member functions must be declared with the class.

member functions may be defined outside of the class, using the scope resolution operator

Member functions need only be declared within their class, they may be defined outside the class, perhaps in a separately compiled module.

To associate function definitions with their class, a new scoping operator is introduced in C++.

on the side

The scoping operator is sometimes called the 'box' operator.

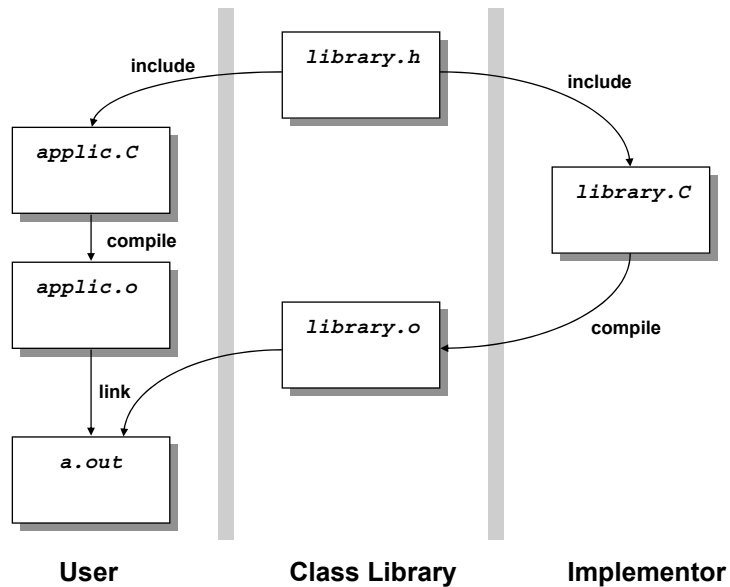
`Time::setTime`

The scope resolution operator (or box operator) binds the function back to the class in which it belongs. Without the operator it would not be possible to determine whether `setTime` belonged to class `Time`, or to some other class with the same function identifier. In fact, without the scoping operator, it would be considered a global function.



Note that default values are part of the interface of a function and should therefore appear with the function declaration in the class, and not with the definition. This informs users that default values exist, and informs the compiler that the reduced parameter list is valid.

Organisation of C++ Programs



C++ programming is based around the production of class libraries to solve problems. Rather than writing a program to solve the problem directly, in C++ one writes (or re-uses) a class library to provide the objects of the problem. The program then models the problem by making objects and establishing their interactions.

When writing a class library, the header file contains the class definition and member function declarations. The members file contains member function definitions and includes the header file. The members file is compiled to produce an object file or library, which when shipped with the header is the C++ class library.

When using a class library, include the class header into the application program, compile to object, and link with the class library to produce an executable.

In C++ it is not envisaged that the implementor of the library should be the user of the library. Consequently, the class header file is said to establish a contract between implementor and user. The implementor may extend or even re-write the members file. As long as the header remains the same, then the contract between user and implementor remains valid.

Organisation of C++ Programs

main.C

```
#include <Time.h>

int main()
{
    Time start;

    start.setTime(1,2,3);
    start.display();
}
```

Time.h

```
class Time
{
    int min, hour, sec;
public:
    void setTime (int, int, int);
    void display ();
};
```

Time.C

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

void Time::setTime(int m,int h,int s)
{
    min = m; hour = h; sec = s;
}

void Time::display()
{
    std::cout << hour << ":"
               << min << ":"
               << sec << std::endl;
}
```

`Time.h` is included into both of the `.C` modules. The `.C` modules are separately compiled and linked together to form an executable.

In `Time.C` the scope resolution operator binds the functions `setTime` and `display` to class `Time`. In `main.C`, the `.` operator indicates that the messages `setTime` and `display` are being sent to objects of class `Time`. It is therefore straightforward for the compiler to determine which functions should be called.

Note that `setTime` and `display` are able to access the private members of `Time`. This is because these functions are members of `Time` and, as such, are part of the implementation.

Member Function Inlining

- Member functions defined in the class body are implicitly inlined

```
#include <iostream>

class Time
{
    int hour;
public:
    void setTime (int);
    void display () { std::cout << hour; }
};

inline void Time::setTime (int h)
{
    hour = h;
}
```

← inlined implicitly

← inlined explicitly

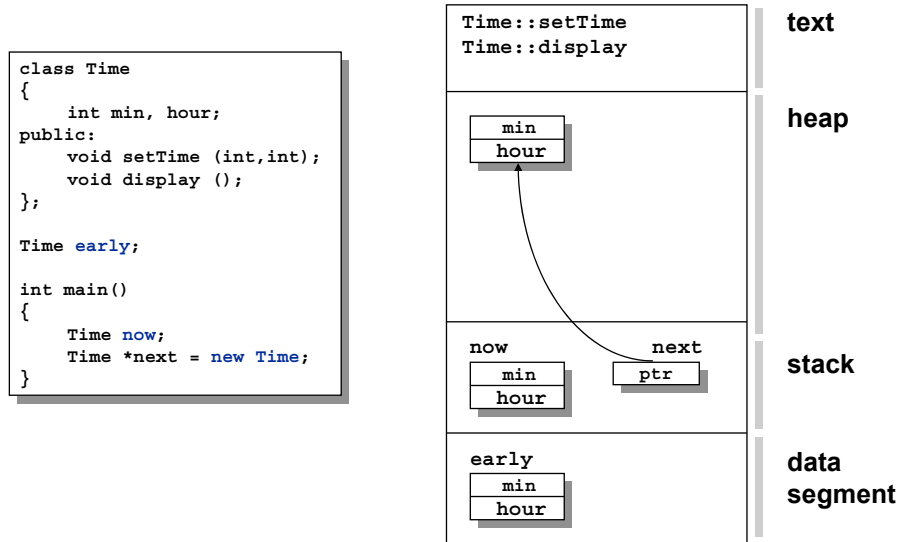
- To inline functions defined in a members file, the file must be included in the header

In the interest of encapsulation users are rarely able to access data members of objects. Updating a simple data member with a new value usually requires the use of an access function. To avoid excess performance overhead, such functions (provided they are defined within the class definition) are implicitly inlined.

Functions only declared in the class, but defined within the header, may be explicitly inlined with the `inline` directive.

As indicated earlier in this class, the determination of whether a function is inlined is at the discretion of the compiler. Most compilers will not inline if the code is too large, unavailable as source or recursive.

Object Memory Organisation



Conceptually objects contain both state and operations. In reality, objects only contain state because the common operations exist once in the text segment of the program. Replicating code with each new instance of an object would be prohibitively expensive.

In the above example three objects are created, a dynamic object on the heap, an automatic object on the stack frame, and a static (global) object in the data segment of the program. Note that a pointer was also automatically allocated on the stack to refer to the dynamic heap object.

This picture is somewhat stylistic. The actual layout on the operating system but in Solaris the order of the segments (from top to bottom) is stack, heap, data and text.

The 'this' Pointer

- When a member function is invoked, it receives a pointer to the object receiving the message
- This hidden first argument is known as the 'this' pointer, and is automatically provided when the method is invoked

note

The code in blue italics is not valid, it is what the compiler effectively generates

```
void setTime (int, int)
void Time::setTime (Time *this, int, int)

void display ()
void Time::display(Time *this)

Time now
now.setTime (1, 2)
Time::setTime (&now, 1, 2)

now.display ()
Time::display (&now)
```

Since member functions are not replicated with each object of a class, some means must be provided of associating member functions with the objects requiring their use. This is achieved by means of a first, hidden, argument, automatically added to member functions.

The examples above show how the compiler maps objects to the required member functions. Although this can sometimes be ignored as implementation detail, explicit reference to the `this` pointer (or at least, knowledge of its existence) is often required when programming.

The most common place of its use is where one object needs to determine if it is acting upon itself. An example occurs when overloading the assignment operator. In order to avoid self-assignment, the left operand must determine if the right operand is the same.

Friend Functions

- Sometimes the private access region can be too strict, and the public access too weak
- In such cases, non-member functions or classes may be declared as friends

```
class Hidden
{
    int secret;
public:
    void set (int s) { secret = s; }
    friend int agent (Hidden *);
};

int agent (Hidden *h)
{
    return h->secret;
}
```

friend declaration

No this pointer, object is passed explicitly

friend functions have nothing directly to do with the class which honours them with friendship. They are not member functions, do not receive the implicit `this` pointer, but gain through the friend declaration, unrestricted access to all members of the class.

The notion of friend functions has generated some controversy in the object-oriented community. They are usually slated as breaking the rules of encapsulation, and thereby increasing coupling and program inter-dependencies. However, there are some instances where using friends saves having to place members in the public interface, and thereby reduces coupling.

friends should only be used when there is a natural coupling between two or more classes which cannot be modelled in any other way. In general they should be avoided. In C++ they are most commonly used when overloading operators to gain symmetry.

Note that friendship is neither transitive nor commutative. Just because John's a friend of Jack, doesn't make Jack a friend of John. Moreover, if John's a friend of Jack, and Jack's a friend of Jill, it does not follow that John's a friend of Jill. *There seems to be a lesson here in human relationships!*

Friend Functions

- class and class member friends

```
class List;
class Node
{
    Node *next;
    Element *ptr;
    friend class List;
};
class List
{
    Node *head;
public:
    add (Element *);
    Element * rem ();
};
```

forward declaration of List

All List functions are friends

The diagram shows a code block with two classes. The first class is a forward declaration of `List`. The second class is `Node`, which contains private attributes `Node *next` and `Element *ptr`, and a friend declaration `friend class List;`. The third class is `List`, which contains a private attribute `Node *head` and two public member functions: `add (Element *)` and `Element * rem ()`. Two arrows point from text labels to the code: one points to the forward declaration of `List`, and the other points to the `friend class List;` line in the `Node` class.

The example shows how one class can make friends of all of the member functions of another class. The purpose here is to enable the `add` and `rem` methods of `List` to be able to access the private attributes of `Node`. The reason why the attributes are not public or that access methods have not been provided, is that `Node` does not wish any parts of the program besides `List` objects to be able to gain access.

A finer degree of friend access control is possible in `Node`. Instead of making all members of `List` friends, `Node` may select specific ones:

```
friend class List::add (Element *);
```

The forward declaration is necessary since `List` is defined after `Node`. Swapping the two around in the file will not help because `List` makes use of `Node`, and `Node` makes use of `List`.

Class Scoping Rules

```
#include <iostream>

int x = 10;

class Scope
{
    int x;
public:
    void display() {
        int x = 0;
        std::cout << x << std::endl;           // local
        std::cout << Scope::x << std::endl;    // object
        std::cout << this->x << std::endl;      // object
        std::cout << ::x << std::endl;         // file
    }
};

int main()
{
    int x = 15;
    std::cout << x << std::endl;               // local
    std::cout << ::x << std::endl;             // file
}
```

The scope rules for C++ are similar to C with the exception of class member functions. Most C functions look for the identifier within the current block, enclosing block or file. C++ member functions look in the current block, enclosing block, class and then file scope.

The C++ scoping operator may be used in a unary form to indicate file scope. Notice also, that in the above example either the `this` pointer or the binary scope operator could be used to resolve to class scope. However, this scope operator is preferred since it is more explicit.

Chapter 11: Constructors & Destructors

- **Initialisation versus Assignment**
 - **Initialising Objects**
 - **Constructor Functions**
 - **Overloading Constructors**
 - **Constant Objects**
 - **Constructing Object Arrays**
 - **Destructor Functions**
 - **Defining a Destructor**
 - **Stack Frame Evaluation**
 - **Dynamic Arrays & Destructors**
-

Initialisation versus Assignment

- Initialisation and assignment are semantically different
- Initialisation occurs when an object is made

```
int i = 5;           // introduces i with value 5
float pi = 3.1415;   // introduces pi with value 3.1415
static int p;        // introduces p auto-initialised to 0
char *ptr;           // introduces ptr uninitialised
```

- Assignment is a method applied to an existing object

```
i = 5;              // changes the value of i to 5
```

There is an important distinction in C++ between initialisation and assignment. The distinction is less important when programming in C because C lacks constructor and destructor functions.

Initialisation occurs when an object is first introduced into a program. Semantically, the object never exists without the initial value.

Assignment is simply an operation applied to an object to ask it to change its value. Assignment occurs after the object exists.

Initialising Objects

```
const int i = 5;  
// const char c;  
int& ir = i;  
// char& cr;
```

Reference and constant objects must be initialised

- **How can user-defined types be initialised?**
 - provide a constructor function
 - implicitly invoked when object created
 - may be overloaded
 - default constructor provided if none defined by user

The semantic difference between initialisation and assignment is demonstrated by constant and reference objects. It is not possible to assign values to either of these, both must be initialised.

Since user-defined types should integrate into the language along side in-built types, there must be some means of initialising them.

The use of member functions, such as `set_val()`, to initialise an object is inelegant and error prone. Because the compiler cannot force the use of the function, a programmer can forget to do so, or (perhaps worse) do so twice. A better approach is to allow the programmer to declare a function with the explicit purpose of initialising objects, which is then implicitly called by the compiler when new objects are introduced. Such functions are responsible for the construction of the object, and are consequently called constructors.

Constructor Functions

```
class Switch
{
    int state;
public:
    Switch (int s)
    {
        state = s;
    }
};

int main()
{
    Switch light = 1;
    Switch alarm (0);

    Switch *ptr = new Switch (1);

    // Switch phone;
}
```

int constructor

constructor may not be called explicitly, there is no return value

both call int constructor

no appropriate constructor

Constructors are member functions which are automatically called when objects of the class are created. Constructor functions have the same identifier as their class.

Constructors take any number of any type of arguments and may be overloaded. The compiler selects the appropriate constructor function based on the number and types of arguments provided when an object is created. Constructors cannot be called explicitly and consequently have no return type.

If no constructors are defined then the compiler provides a void or default constructor. This leaves the new objects state uninitialised. If even one constructor is defined, then the default void constructor is not generated. In the example above, the phone `Switch` object cannot be created because there is no suitable void construction. Where appropriate, class designers always provide their own default (or void) constructor.

Overloading Constructors

```
class Point
{
    int x,y;
public:
    Point();
    Point(int);
    Point(int,int);
};

Point::Point() { x=0; y=0; }
Point::Point(int a) { x=a; y=0; }
Point::Point(int a, int b) { x=a; y=b; }

int main()
{
    Point origin;
    Point start (4);
    Point stop (8,2);
    Point mid = 6;
    Point copy (start);
    Point p();
}
```



In the above, three constructors are provided. The first four objects are initialised by calls to the appropriate constructor. Note that when using the single argument constructor, two forms are possible



```
Point start (4);
Point mid = 6;
```

In general the former is preferred since it is consistent with the other initialisations, and avoids the syntactic similarity between initialisation and assignment.

In the above, the copy object is constructed from another compiler provided constructor. The default 'copy constructor' initialises an object by performing a member-wise copy from another object of the same type. Generally, programmers provide their own copy constructors because member-wise copy is inappropriate for objects with pointer members.

The last construction above is not a construction at all. In fact, it is a declaration for a function which takes no arguments but returns a `Point` by value. Care should be taken to avoid confusion here. To use the void constructor, simply specify an identifier for the object.

```
Point dot;
```

Constant Objects

constant objects must be initialised, they cannot be assigned new values

```
class Point
{
    int x,y;
public:
    Point(int = 0,int = 0);
    void set (int, int);
    void display () const;
};

Point::Point(int a, int b) { x=a; y=b; }
Point::set(int a, int b) { x=a; y=b; }
Point::display() const {
    cout << x << y << std::endl;
}

int main()
{
    const Point origin;
    Point current (origin);

    origin.display();
    current.display();
    // origin.set (10,10);
    current.set (10,10);
}
```

Just as with intrinsic types, when constant objects are made from user-defined types, their state cannot be changed.

Therefore, when instantiating constant objects appropriate constructor values should be provided. In the example above, the `origin` object uses the only constructor available and accepts the two default arguments.

The only functions permitted by constant objects are those which do not change their state. To indicate that a member functions will not change the objects state, place the keyword `const` after the parameter list in both the declaration and definition.

```
void display() const;
```

`const` is actually being applied to the first hidden argument of the member function, namely the `this` pointer. The compiler interprets the declaration as

```
void Point::display (const Point *this)
```

Note in the above class definition, that the three overloaded constructors on the previous page are replaced with one constructor with default values. This demonstrates a useful technique, but take care to avoid ambiguities if using both overloaded functions and default values.

Constructing Object Arrays

- static and auto arrays can take initialisation values

```
Point array[] = { Point (1,2), Point (4,2), Point (2) };
```

- dynamic arrays use void construction

```
Point *ptr = new Point [3];  
ptr[0].set (1,2);  
ptr[1].set (4,2);  
ptr[2].set (2);
```

← A void constructor
must exist

Unfortunately, the syntax used when providing constructor values for automatic or static arrays of objects does not extend to dynamic arrays. Consider the following

```
Point *p = new Point [3] = {Point (1,2), Point (1)};
```

The first = is for assignment and the second for initialisation. It is not surprising that this is not legal code. Note that it is possible, however, to initialise single dynamic objects.

```
Point *p = new Point (3,4);
```

Since constructor values cannot be provided `new` must call a void constructor for each object in the array. If a void constructor is not available, then the code will not compile.

It may seem necessary to always provide a void constructor so that programmers can use dynamic arrays of the objects of your class. Whilst it is good not to exclude this usage, it is important that the provision of a void constructor (directly or with default arguments) only be done if it is semantically correct for the class. For example, it may not be sensible to allow a void construction of a bank account object, if your design requires that all bank account objects be associated with an account holder (Person or Company, say).

Dynamic arrays of objects are inflexible since they cannot be extended, inserted, sorted, contracted. In practice, the structure is rarely used for anything beyond simple (and often intrinsic) types.



Destructor Function

- The destructor function cleans up when an object dies
- Only one destructor may be provided
 - called implicitly when object dies
 - no return type
 - no function arguments
 - named after class with ~ prefix
- Allows controlled clean-up
 - generally, undoes the construction
 - deletes dynamically allocated memory
 - closes files
 - collapses windows

The destructor function is automatically (and implicitly) called when an object dies. In the case of an automatic variable, this is when the function returns; in the case of a static, it is when the program terminates; and in the case of a dynamic variable, when it is deleted.

Destructors return resources claimed by the object when it was created or during its lifetime. Destructors are essential because the implementation of an object is hidden from users of the object. A user has no idea whether further memory was allocated, files were opened, or communication sockets with another process established. Consequently the user cannot be expected to manage these resources. Since the destructor is implicitly called the burden of resource management is placed squarely on the shoulders of the class designer.



Note that the destructor does not delete the object. Memory allocated for the object is returned according to the storage class of the object. The destructor merely cleans-up before this occurs.

Defining a Destructor

```
#include <cstring>

class String
{
    char *str;
public:
    String (const char *s = "")
    {
        str = new char [std::strlen(s) + 1];
        std::strcpy (str, s);
    }
    ~String ()
    {
        delete [] str;
    }
};

String name ("fred");
```

the constructor is used to claim memory for the character array.

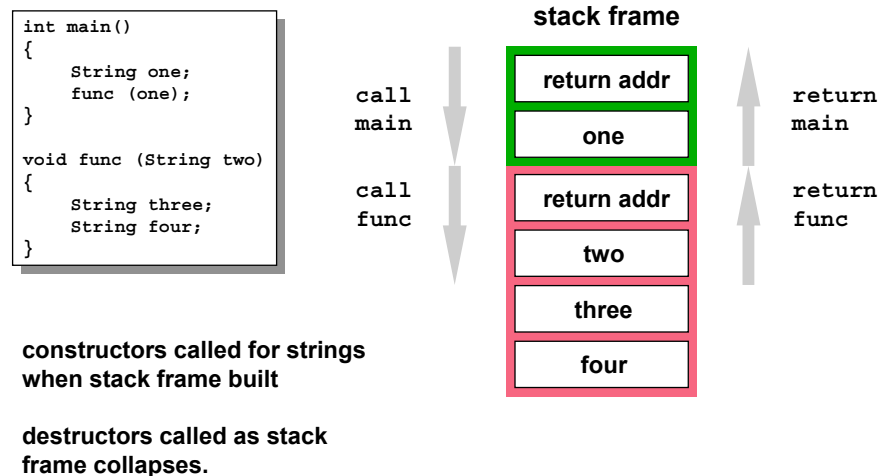
the destructor is used to return the memory when the object dies.



When an object of `String` is instantiated, space for the object is first claimed, and then the constructor is called. This sets up the object and allocates any further memory which may be required. The destructor usually undoes the work of the constructor. It is called automatically when the object dies to clean up.

Note that the destructor does not release the memory allocated to the `String` object. It merely releases any further space associated by the object during its lifetime. In this example the `String`'s constructor allocates a dynamic array of characters and stores the address in the `String` object. The destructor simply ensures that this array is returned to the heap before the object's memory (which, of course, contains the array's address) is released.

Stack Frame Evaluation



When functions are called, actual parameters and a return address are placed on the stack. The stack grows as functions are entered and shrinks when functions exit. The portion of the stack associated with a function is referred to as its 'stack frame'.

The rather simplified picture above shows that the first item on the stack frame is the return address. This is the address to return control when the function exists. Next on the stack frame are the actual parameters passed into the function, and then local variables belonging to the function.

Creating an Object

When an object is created the necessary memory is claimed in accordance with its storage class. In the example above, it is claimed from the stack. Next, the object's constructor is called to initialise the memory. In the case of the `String` class, this requires that further memory be claimed from the heap.

Deleting auto objects

When the function exits, control is passed back to the calling routing (in accordance with the return address). At this point the stack frame is collapsed. First, the destructors for each of the objects on the stack frame are executed. The `String` destructor frees the heap memory claimed during construction. Finally, the space reserved for the objects on the stack is released.

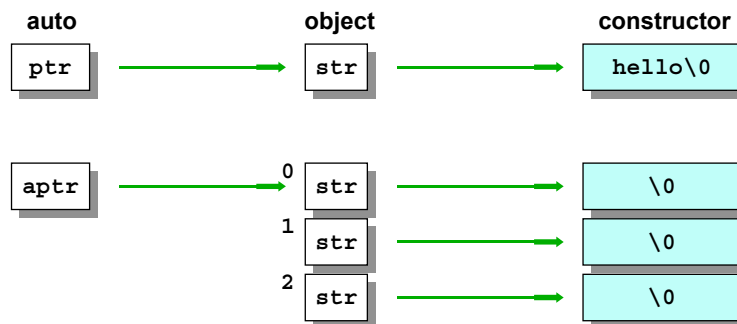
Note that the code above is seriously flawed. When object one was passed into the function the default copy constructor was used to initialise object two. This is insufficient when objects contain pointers and is explained in depth later in the course.



Dynamic Arrays & Destructors

```
int main()
{
    String *ptr = new String ("hello");
    String *aptr = new String [3];

    delete ptr;
    delete [] aptr;
}
```



When the single object was allocated on the heap, memory was claimed for the object proper (i.e., space for the member `str`), then control was transferred to the constructor. This allocated further memory to hold the `char` array and stored the address in `str` in the object. The constructor then returned control to `new`, which in turn returned the address of the object for assignment to `ptr`.

When the array of objects was allocated, `new` first claimed space for all of the objects (i.e., enough space for the member `str` in each of the objects). The void constructor was then called for each object (recall that `void` is the only constructor which can be called with dynamic arrays). The constructors allocated further memory and stored the addresses in each object's `str` member. The address of the first object was then returned by `new` for assignment to `aptr`.

When `delete` was called on `ptr`, control was first passed to the destructor. This released the extra space allocated by the constructor which contained the `char` array 'hello'. Then `delete` released the space for the object proper.

When `delete []` was called on `aptr`, the `[]` informed `delete` that `aptr` addressed an array of objects. `delete` subsequently called the destructor for each object, and then returned the memory they occupied to the heap.

Failure to specify the `[]` when deleting an array of objects has undefined behaviour. However, in general it causes the destructor on only the first object to be called, and then the memory for all the objects to be released.



Course Notes

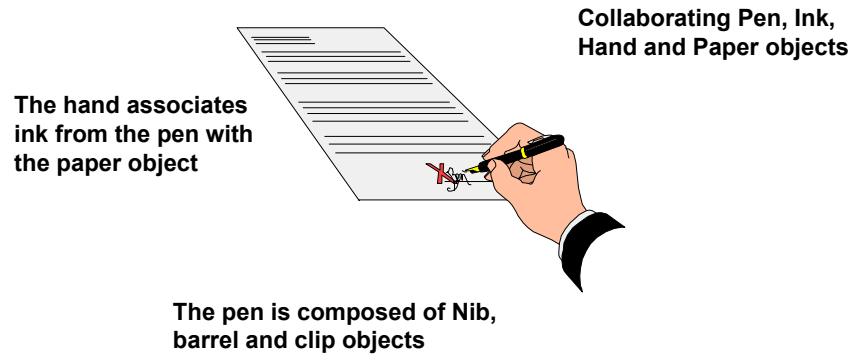


Chapter 12: Object Modelling

- Object Collaboration
- Composition
- Member Access
- Construction
- Association
- Associative Custody
- Composition versus Association

Object Collaboration

- **Objects contribute to the overall behaviour of a system by collaborating with each other**



In an object-oriented system objects co-operate to solve the problem. There are two main relationships which exist between objects: composition and association.

Composition is the relationship of containment. In the above example, the Pen is composed of nib, ink, and cylinder objects. Association is the weaker relationship of uses. One object uses another for a period and then disassociates itself from it. In the above example the Hand object uses the Pen object and the Paper object to support the system 'signing a letter'.

Object Composition

- **Defining objects within objects**
 - called aggregation in C
 - class members are other class objects
 - the OO “HAS-A” relationship
- **Objects have intersecting lifetimes**
 - space is allocated for the composite object and all components
 - constructors and destructors are called for all objects
- **Composite object owns components**
 - composite is responsible for components
 - must ensure proper construction
 - provides interface functions to hide details of components

Composition promotes re-use, and thereby may raise the productivity and reliability of the software system. In addition, composition increases the level of abstraction at which the class designer must work.

Object Composition

```
#include <iostream>
#include <string>

class Date
{
    int day, mon, year;
public:
    Date (int d=0, int m=0, int y=0) {
        day=d; mon=m; year=y;
    }
    friend ostream& operator << (ostream& o, const Date& d) {
        o << d.day << "/" << d.mon << "/" << d.year << endl;
    }
};

class Employee
{
    Date start;
    std::string name;
public:
    Employee (int d, int m, int y, std::string n);
    void display() const {
        cout << "Employee: " << name << " " << start << endl;
    }
};
```

An **Employee** object is composed of a **Date** object (holding the **Employee's** start date) and a **String** object (holding the employee's name).

It is usual for the components to be hidden and for the composing objects to provide interface functions onto them where necessary. This is part of encapsulation; users of **Employee** objects should not need to know what **Employee** objects are made of. Such knowledge is implementation specific, and if available to users may increase the coupling in the system.

Object Composition

- How are composite objects initialised ?



```
#include <Employee.h>

Employee::Employee (int d, int m, int y, std::string n)
{
    name = n;
    start = Date (d, m, y);
}
```

This is assignment, not initialisation!

Both name and start have already been initialised using their void constructors

The above constructor for `Employee` does not initialise the component objects. The two statements in the body assign new values to `name` and `start`. This is assignment because the two objects exist and the statements use the assignment operator.

There is no mechanism to initialise the components inside the constructor body, primarily because the components already exist. They exist because they are part of the composite object, and the composite exists.

The above solution is flawed in a number of ways. First, both `name` and `start` have already had their void constructors called. If they do not have void constructors, then the program will not compile. Second, both components must have a public assignment operator available, and this may not always be true. Third, the process is inefficient. For example, to set-up `start` an anonymous `Date` object is created on the stack frame of the constructor.

Constructing Composites

- **Composite must pass values to initialise the members**
 - each component is then responsible for initialising itself

```
class::class (parameter list) : initialisation list
{
    constructor body
}
```

- **Initialisation list**
 - enables values to be passed to constructors
 - any member may be initialised in this way
- **Execution body**
 - can only 'assign' values to members and is often empty

Class constructor functions are split into two parts. The first is an initialisation list which allows values to be passed to constructors of composite members. The second is a body (which is often empty) used to set-up the composing object.



The semantic difference between the initialisation list and the constructor body is the same as that between normal variable initialisation and assignment. Only the former can be used to give values to constants, references, and other constructors.

Constructing Composites

on the side

Initialisation values must be passed to all components that are expecting a value.

```
#include <date.h>
#include <string>

class Employee
{
    Date start;
    std::string name;
public:
    Employee (int, int, int, std::string);
    void display() const;
};
```

employee.h

```
#include <employee.h>

Employee::Employee (int d, int m, int y, std::string n)
: start (d,m,y), name (n) {}
```

employee.C

↑
each argument is
an expression

The initialisation list is a comma separated list of initialisers between the : and function body {}.

Each element in the list is the name of the object being initialised and a set of arguments to be passed to the object's constructor. Note that each argument (as with standard function calls) is an expression. It may therefore hold a constant value, be derived from other parameters, or even be the result of calling some function.

Failure to properly construct the component objects will cause a compilation error. Where a component has a constructor, values must be supplied to ensure its proper initialisation.

In the above example, however, both the `Date` and `String` component classes have default constructors. Therefore, in this instance it would have been possible to make `Employee` objects without explicitly initialising the component objects.

Construction / Destruction Order

- **Construction**

- member constructors execute first
- construction occurs in order of declaration
- initialisation list order is irrelevant
- composite object is constructed last

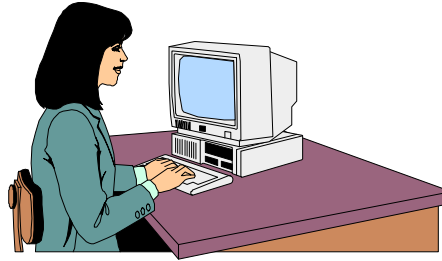
- **Destruction**

- lifetime of the objects is bound together
- composite destructor is called first
- component destructors are called in reverse order of declaration

The construction and destruction order is specified in the language definition. The rigid ordering is important because of possible dependencies between class members. For example, one member may be initialised to a value dependent on the value of a previous initialised member.

Association

- Independent objects cooperate to achieve the goal



the **Person** object **USES**
the **Computer** object to
complete a task

Association is a weaker relationship than composition. The objects being related don't have to be created or destroyed at the same time, and are not intimately bound together.

In the above example, the **Person** object uses the **Computer** object to solve some problem. The keyboard and display of the **Computer** object represent the public interface. The fingers of the **Person** object are referential attributes, and are the means by which messages are conveyed to the **Computer** object.

Association

- **Implemented through pointers & references**
 - objects maintain pointers to other objects
 - the OO “USES” relationship
- **Objects need not have intersecting lifetimes**
 - objects may be created independently
- **Ownership is flexible**
 - an object may be created by one class
 - and destroyed by another
 - more work to be done by client code
 - custody must be explicitly managed

Association is much more flexible than composition at the expense of increased work for the programmer. Specifically, each of the objects in the relationship must be created, associated at the relevant time with co-operating objects, generally managed, and finally destroyed. That is, custody must be explicitly managed.

Association is flexible because it allows one object to be associated with many others during its lifetime. An object may even have many associates simultaneously. In the case of composition, it is not possible for an object to be composed within more than one other object.

Association

```
class Book
{
    std::string title;
    int isbn;
public:
    Book (std::string, int);
    ~Book ();
    void display () const;
};

class Library
{
    Book *shelf[100];
public:
    Library();
    ~Library();
    void add (const Book&);
    void display() const;
};
```

**The library is associated
with Book objects**

**custody must be
explicitly managed**

The books and the library do not have intersecting lifetimes. It is unlikely that the library creates the books, but it probably takes custody of them. Custody means having responsibility to destroy the books (perhaps when they are damaged).

Constant & Reference Members

```
class Data
{
    int i;
    const float f;
    int & r;
public:
    Data (int, float, int&);
    void display() const {
        std::cout << i << f << r << std::endl;
    }
};
```

data.h

```
#include <data.h>

Data::Data (int a, float b, int& c)
: i(a), f(b), r(c) {}
```

data.C

```
#include <data.h>

int main() {
    int num = 101;
    Data test (4, 93.8, num);
}
```

main.C

The initialisation list must also be used when classes contain constant or reference members.

The constant `f` is initialised when objects of `Data` are created, and may not be changed for the lifetime of the objects. The reference `r` must also be initialised when the object is created, and (since it is essentially a constant pointer) may not be subsequently changed.

Constant Associations

Reference members provide a mechanism to implement constant associations. The objects `num` and `test` are created at different times but `num` is associated with `test` for the duration of `test`'s lifetime.

Association versus Composition

- **Composition**
 - management done automatically
 - lifetimes locked
 - inflexible
 - implements OO “has -a”

- **Association**
 - management done manually
 - lifetimes independent
 - flexible
 - implements OO “uses”

Composition (sometimes termed aggregation) and association are two building blocks for relating objects in systems.

Often, a hybrid of composition and association is implemented. Association can be made to look like composition by hiding the allocation of new objects behind member functions, and managing the custody of the ‘association-component’ within the enclosing object.

Course Notes



Chapter 13: Class Data & Functions

- Static Data Members
- Static Member Functions
- Shallow & Deep Copy
- Providing a Copy Constructor
- Conversion Constructors

Class Data & Functions

- **Class members belong to the class**
 - not to individual objects of the class
- **Class members may be accessed at any time**
 - even before objects of the class exist
- **Class members implement the ‘meta-class’ concept**
 - class (static) data provides shared data to objects
 - class (static) functions access shared data
 - constructor functions initialise objects
 - destructor functions clean-up objects

Class data members and function members belong to the class in the sense that they do not occur within any individual object. A class essentially provides a blue-print for the objects which are made from the class. Class members are not part of this blue-print. Rather they provide the class with its own behaviour and its own state. Class members are like the classes class; and the class is a kind of instance of this class. This notion is what underpins the ‘meta-class’ concept in object-oriented systems.

Class members are useful because they allow the class to be involved in the system before objects of the class have actually been created. They also provide a useful repository for data which is common across all objects of the class.

Constructors and destructors are like class functions in that they operate somewhat outside of the normal behaviour of an object.

Static Data Members

- **Class variables**
 - reflect the state of the class not individual objects
- **Only one copy of the data**
 - stored in the program's data segment
- **Visible from all objects of the class**
 - class scope
 - global to all objects of the class
- **Useful for reference counting**
 - counting the number of instances

Static data members are class wide global data. The data exist once for each class (rather than appearing with each instance) and resides in the program's data segment.

Static data members are useful when the existence or quantity of an object needs to be determined

Static Data Members

```

class Count
{
    static int inst;
public:
    Count () { ++inst; }
    ~Count() { --inst; }
    int get() { return inst; }
};

int Count::inst = 0;

int main()
{
    Count a, b, c;
    std::cout << "instances = " << a.get() << std::endl;
    Count d, e;
    std::cout << "instances = " << d.get() << std::endl;
    Count *p = new Count[4];
    std::cout << "instances = " << p->get() << std::endl;
    delete [] p;
    std::cout << "instances = " << e.get() << std::endl;
}

```

declaration

definition (one off in members file)

instances = 3
instances = 5
instances = 9
instances = 5

Note that the entry in the class definition is only a declaration of the member. When objects of class `Count` are created, the static member is not present. The static member must be defined (thereby allocating space) elsewhere in the program. The definition may only occur once, and is usually placed in the class members (implementation) file.

Static data members can be accessed just like any other members. When private only member functions and friends gain access, when public users of objects gain access.

Since static data members belong to the class rather than particular objects, they tend to be identified with the scope resolution operator.



```

Count obj;

obj.inst;           // this is possible
Count::inst;        // this is preferred.

```

This form is preferred because it shows the binding with the class more clearly.

Static data defined for member functions is also static for the class. Therefore, such data appears only once for the class, not with each function.



Note from the above example, that instance counting may sometimes generate unexpected results. Anonymous objects can be created and destroyed during expression evaluation, and as a consequence of passing parameters into and out of functions.

Static Member Functions

- **Static member functions belong to the class**
 - can only access static (class wide) data
- **Static member functions do not get the ‘this’ pointer**
 - cannot call non-static member functions
 - can only access object members explicitly
- **Invoked through the class**

```
class::function()
```

Static member functions, like static data, belong to the class. The use of static member functions allows the compiler to verify that the function only accesses static data. Consequently, the function may be invoked from class qualification, rather than through an object.

Static Member Functions

```
class Count
{
    int x;
    static int inst;
public:
    Count (int val) : x(val) { ++inst; }
    ~Count() { --inst; }
    void print() const {
        std::cout << x << " / " << inst << std::endl;
    }
    static int get();
};

int Count::get() {
    // ++x;           // illegal not static
    return inst;      // return Count::inst
}

int Count::inst = 0;

int main()
{
    Count a(10);
    std::cout << "static = " << Count::get() << std::endl;
    a.print();
}
```

**static = 1
10 / 1**

Static member functions are useful when it is necessary to query the state of a class (or collection of objects).


Since the functions may be called using the class name, rather than an instance name, they can be invoked before any objects exist. For example, to determine whether there are any instances at this time.

Shallow Copy

- All classes are given with a default copy constructor
 - performs a shallow (member-by-member) copy

```
class Point
{
    int x,y;
public
    Point(int a, int b) : x(a), y(b) {}
    void display() const {
        std::cout << x << ", " << y << std::endl;
    }
};

int main()
{
    Point origin (0,0);
    Point start (origin);
    start.display();
}
```



calls default copy constructor

A default copy constructor is provided by the compiler for all classes, unless you provide a copy constructor yourself. The default constructor implements member-wise copy, or 'shallow copy'. The values of members from one object are copied to aid the construction of another object.

In the example above, the object `start` is constructed from the object `origin`. This caused the default copy constructor to copy the values of the members of `origin` to `start`.

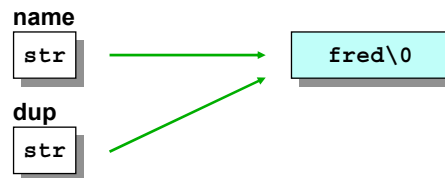
Shallow Copy

shallow copy is unsuitable when the class contains pointers

how many times is the memory released?

```
class String
{
    char *str;
public:
    String (const char *s = "") {
        str = new char [std::strlen(s) + 1];
        std::strcpy (str, s);
    }
    ~String () { delete [] str; }
    void print () const {
        std::cout << str;
    }
};

String name ("fred");
String dup (name);
```



Shallow copy is unsuitable when the class contains pointers. This is because the value of the pointer member (which is all that is copied) is an address. Thus two objects will contain members which address the same part of memory.

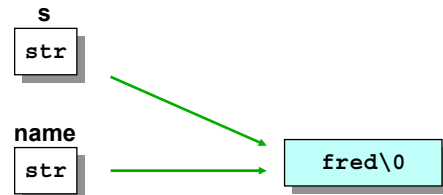
In the example above, the first `String` object is initialised using the user-defined `const char*` constructor (note the `const` means that the constructor may be called with `const` or non-`const` character arrays). A second `String` object (`dup`) is then initialised from the first using the default copy constructor. Shallow copy leaves both objects pointing to the same part of memory.

The above situation is dangerous. The encapsulation of private data has been destroyed. When the `name` object is changed a corresponding change (side-effect) occurs to `dup`. `name` will have changed the state of `dup` without going through its formal public interface. Moreover, the first object to die will (through its destructor) cause the `char*` array to be deleted. Consequently, when the second object dereferences the pointer (or even attempts to die) a memory fault will occur.

Implicit Initialisation

```
void print (const String s)
{
    s.print();
}

int main()
{
    String name ("fred");
    print (name);
}
```



Pass by value causes implicit initialisation of local function arguments.

What happens to the local 's' variable when the function returns and its stack frame collapses?

Not all initialisations are as explicit as the those on the previous page. Passing function parameters by value also requires use of a constructor. Actual parameters are created on the stack frame for the function.

In the above example a `String` object is passed into the function `print`. Since pass-by-value is employed a copy of the original `String` (made using the default copy constructor) is created. The function declares that it will not change the `String` object and simply calls a passive function object (note, the `display` member function must be declared `const`).

Although the program appears to be innocuous, it will generate a run-time error. On calling the function the value of the `str` member for the object on the stack frame is the same as that of the original object `name`. This is the nature of shallow copy. On return from the function the stack frame will collapse and the local copy (`s`) will die. This in turn will cause the destructor to release the (common) memory address by `s.str`. Consequently, the `name` variable (apparently passed into a passive function) will have become corrupt!

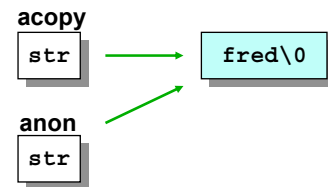
This situation is avoided by providing an appropriate copy constructor. Alternatively, the actual parameter should have taken a reference to the `name` object. This is more efficient, and only involves the creation of a new pointer not a new `String` object.



Implicit Initialisation

```
String input ()
{
    char temp[100];
    std::cout << "please enter string "
    std::cin >> std::setw (100) >> temp;
    String acopy (temp);
    return acopy;
}

int main()
{
    String one = input();
}
```



Return by value causes
an implicit initialisation

An anonymous variable is
initialised with the object
returned from the function

Implicit initialisation also occurs when functions return by value. The value being returned is copied from the functions stack frame to an anonymous variable in the calling expression. Once the stack frame has collapsed, the anonymous object in the expression holds a pointer to memory no longer allocated.

This is particularly subtle, since not only is the construction implicit, the actual existence of the object is not apparent -- it is anonymous!

In the above example it appears that the compiler generates code first to copy the new object to the calling expression, and then to the constructor for the one object. Since corresponding destructors would also have to be called, the code generated would be large and inefficient. Consequently, compilers are allowed to optimise over the particular number of constructor/destructor pairs which need be called when evaluating expressions. Call Named Return Value (NRV) and requires a copy constructor to be defined---though it is not used.

Stroustrup writes ...

"The fundamental rule is that the introduction of a temporary [anonymous] object and the calls of its constructor/destructor pair may be eliminated if the only way the user can detect its elimination or introduction is by observing side effects generated by the constructor or destructor calls."

Annotated Reference Manual, pp299-300

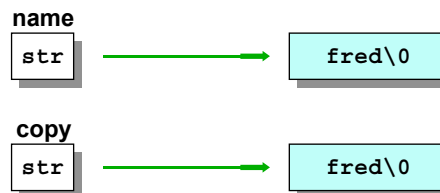
Deep Copy

- Override the default copy constructor for deep copy

```
class String
{
    char *str;
public:
    String (const char *);
    String (const String& s) {
        str = new char [std::strlen (s.str) + 1];
        std::strcpy (str, s.str);
    }
};

String name ("fred");
String copy (name);
```

The constructor
takes a String&



The solution to the copy initialisation problem is to override the default copy constructor. A copy constructor defined for `String`'s should allocate new memory for the character array, and copy each character from the original object.

Note that the copy constructor should take a reference to the class type. Passing by value into the copy constructor would require a copy constructor to be called. Since this is recursive, most C++ compilers raise an error.

It is good programming practice to always provide a copy constructor when the class contains pointers. Even if the current usage of the class does not require copying, future uses (as the program evolves or the class is used in a new context) may do.

You may also define a private copy constructor to prohibit copying except by friend and member functions of the class. Since a constructor now exists the compiler will not supply its default.



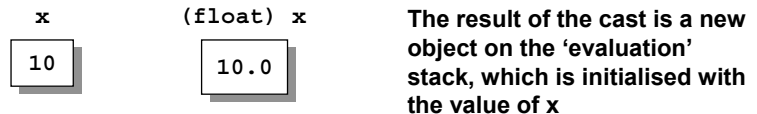
Type Conversions

- C++ allows implicit & explicit casts between types

```
int x=10, y=5;
float result;

result = x / y;           // implicit cast of (x/y)
result = (float) x / y;   // explicit cast of x
result = float (x) / y;   // functional explicit cast
```

- Casting involves constructing anonymous objects



Type conversions (i.e., casts) are common practice in both C and C++. C++ provides the additional facility of allowing such conversions to involve user-defined types.

When an object of one type is coerced into an object of another, then the original object is used as an initialiser for the other. That is, the original object is not changed, instead a new object of the appropriate type is constructed. The type of the original object dictates which constructor should be used. For example, in the above `int / float` examples, an anonymous `float` object is constructed using its integer constructor.

The ANSI C++ standard brought in a set of casting operators to make casting more explicit. These safer casts are discussed in the Advanced C++ class.

```
static_cast<>
dynamic_cast<>
const_cast<>
```

Constructor Conversions

- Constructors define how to make objects
- Constructor arguments specify type of conversion

```
Data::Data (const char*)
Data::Data (int)
Data::Data (const Point&)
Data::Data (const Time&)
Time::Time (int,int)
Time::Time (std::string)
```

```
std::string one, two;

two = (std::string) "hello";    // explicit cast
one = "hello";                 // implicit cast

Time t = Time (12, 45);        // functional style cast
one = t;                       // implicit cast from Time
```

Constructor functions provide the means of converting from intrinsic and user-defined types to user-defined types. The compiler calls the constructor most appropriate for the cast.

In the above example, the `char*` `"hello"` is coerced to `String` both implicitly and explicitly. It is subsequently assigned to a `String` object using the default (member-wise) assignment operator.

To enable casts to user-defined (i.e., complex) types, the functional style of cast is used in C++.

```
t = Time (12,45)                // two initialiser values
t = (Time) 12 , 45              // comma operator used!
```

Note that conversion from user-defined types to intrinsic types requires another mechanism. Conversion operators (discussed in the next chapter) are defined for this.

Course Notes



Chapter 14: Operator Overloading

- Dangers & Restrictions
 - Default Operators
 - Operator Functions
 - Overloading Binary Operators
 - Overloading Unary Operators
 - Friend & Member Functions
 - Overloading Assignment
 - Operator Arguments & Return Types
 - Conversion Operators
-

Operator Overloading

- **Allows operators to work with user defined types**
 - classes integrate into the language along with intrinsic types
 - improves readability of code

```
int a=5, b=10, c;  
  
c = a + b;
```

```
Complex a(3,4), b(8,3), c;  
  
c = add (a,b);      // free function  
c = a.add (b);      // member function  
  
c = a + b;          // overloaded +
```

Operator overloading allows user-defined types to be used with the language operators in the same way as intrinsic types. Part of the design rationale for C++ was to integrate user types as first class elements of the language.

When used properly operator overloading increases the readability of the code. However, when used badly, it can render the code incomprehensible.

Dangers & Restrictions

- **Programs may be difficult to read**
 - consider '+' defined to subtract two complex numbers
 - always adhere to the implied semantics of the operation
- **Ensure semantics are sensible**
 - adding two airports does not necessarily make a bigger airport
- **Restrictions**
 - cannot change precedence, associativity or arity
 - cannot introduce new operators
 - cannot redefine operators for intrinsic types
 - the following cannot be overloaded

. . * :: ? :

One of the dangers with operator overloading is that it provides the ability for users to apply their own semantics. Consider overloading + to subtract rather than add! More generally, consider using + to increase an `Employee`'s salary.

Operators should only be overloaded if the use is similar to the way the operator works with intrinsic types. This, however, requires that programmers fully understand (to a greater extent than C) the semantics of the language operators.

Never change the implied semantics of an operation.



Restrictions

Precedence refers to the order of evaluations of an expression

```
a + b * c           // * is evaluated before +
```

Associativity is the direction of evaluation given similar precedence

```
a + b + c + d       // is left associative
```

```
a = b = c = d       // is right associative
```

Arity is the number of arguments

```
a++                // ++ is a unary operator
```

```
a + b              // + is a binary operator
```

Operator Functions

- Operators are functions and have functional names

+	operator+()
-	operator-()
=	operator=()
+=	operator+=()

Note this is a
distinct operator

- The following are syntactically correct

a + b;	
operator+(a,b)	// free function
a.operator+(b)	// member function

- To overload operators, overload the functional names

Operators give rise to functions implemented by the compiler. In addition to the operator symbol, they also have functional names. These functions can be overloaded to provide implementation of operators for user-defined types.

In the second block above, `operator+` has been overloaded for some user-defined type. If it is overloaded as a general function, then both operands (note `+` is binary) must be explicitly supplied. If it is overloaded as a member function, however, then the left hand operand is implicit. It is the object to which the message `operator+` is being directed.

Overloading binary +

```
class Fraction
{
    int numer;
    int denom;
public:
    Fraction (int n=0, int d=1) : numer (n), denom (d) {}
    Fraction operator+ (const Fraction& right) const
    {
        int n = numer * right.denom + right.numer * denom;
        int d = denom * right.denom;
        return Fraction (n,d);
    }
};

int main()
{
    Fraction a(1,2), b(1,4), c;
    c = a + b;
    c = a.operator+(b);           // same as a + b
}
```

`operator+` is defined for the `Fraction` class to enable `Fraction` objects to be added just as `int` or `float` objects are added.

In this implementation `operator+` is a member function. Consequently the left hand operand is `'this'`, and the members can be accessed directly. The right operand is the only explicit parameter, its members are accessed using the `'.'` operator.



The operator argument and return types are important when overloading. Considerations of efficiency and semantics must always be borne in mind. In the above example, a `const&` was supplied because the right hand operand is not changed, and it is more efficient to pass by reference. A reference was not returned because upon completion of the function there would be nothing left to refer to. The anonymous automatic `Fraction` created by the `return` construct dies when the function's stack frame collapses. (This explanation does not take compiler optimisations such as NRV into account).

Overloading ++

on the side

With `operator++()` the object itself is returned, with `operator++(int)` a copy of the object is returned. The latter is essential because the result is being copied off the stack frame.

```
class Date
{
    int day, mon, year;
public:
    Date (int d, int m, int y)
        : day(d), mon(m), year(y) {}
    Date& operator++();
    Date operator++(int);
};

Date& Date::operator++()
{
    ++day;
    return *this;
}

Date Date::operator++(int)
{
    Date temp = *this;
    ++day;
    return temp;
}

int main()
{
    Date today (15,6,94);
    Date tomorrow = ++today;
}
```

prefix ++
postfix ++

Why return by reference?

Why return by value?

Overloading unary operators is similar to binary operators except that there are parameters (besides the implicit `this`). The exception to this rule is for prefix and postfix increment and decrement. To distinguish between prefix and postfix, an anonymous `int` is declared for the postfix operator.

The implementations for both operators are similar. The interesting part of postfix `++` is that the existing values of the object must be returned, but after the calling expression, the object's value should have increased. The `this` pointer is used so that the object can be incremented but its original value returned.

A reference could be returned in the prefix `++` for efficiency. Since the object exists outside of the class, it is safe to refer to it. However, a reference cannot be returned for postfix `++`. The object being returned is temporary and lives on the function's stack frame. Therefore, to preserve its value, it must be copied out of the function before the automatic is destroyed.

There is a subtle semantic distinction between return by reference and by value for the prefix increment. The distinction is that if return-by-reference is used, then the calling expression evaluates to an l-value. Alternatively, if return-by-value is used then the result is only an r-value and may not be subsequently assigned to. On intrinsic types the following is valid, so return by reference is the most suitable.

```
++++i;
```

Overloading > And <

```
class Date
{
    int day, mon, year;
public:
    Date (int d, int m, int y) : day(d), mon(m), year(y) {}
    int operator > (const Date& right) const { return day > right.day; }
    friend int operator < (const Date& left, const Date& right);
};

int operator < (const Date& left, const Date& right)
{
    return left.day < right.day;
}

int main()
{
    Date today (15,6,94), sellby (20,5,80);
    while (today > sellby)
        ++sellby;
}
```

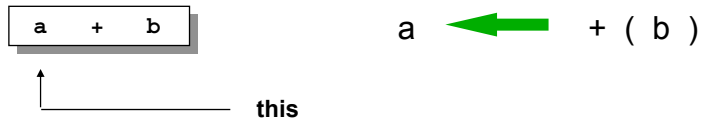
Overload as free function or member ?

Operators may be overloaded as friends or members. Free functions have little to do with the class, but gain access to the private members because they are typically declared friends. The function declaration must be given inside the class, but does not suggest any other coupling. The free functions definition may appear anywhere in the program and is not scoped relative to the class.

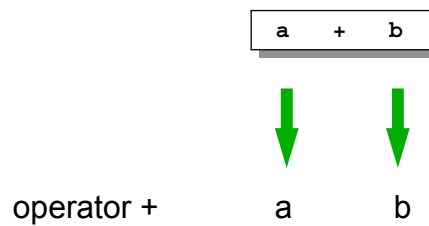
Free functions do not receive the `this` pointer and must therefore be provided explicitly with the arguments operated upon. Unary free functions have one arguments, binary free functions have two.

Free or Member Function

- Member functions have an implied first argument



- Free functions require all arguments to be explicit



There is an important semantic difference when calling a free (usually friend) function as opposed to calling a member function.

Member functions are not directly called. In the above use of a member function, the object `a` receives the message `+ (b)`. This indicates the use of the `operator+` method with the argument `b`. It is significant that `a` is on the left, since `a` becomes the implicit `this`.

Free functions are normal functions as one might find in C. The function is called with two arguments, `a` and `b`. Neither argument is distinguished, and the `'.'` operator must be used to access the members of both.

Operator Member Functions

```
class Fraction
{
    int numer;
    int denom;
public:
    Fraction (int n=0, int d=1);
    Fraction operator+ (const Fraction& right) const;
};

int main()
{
    Fraction a,b,c;
    c = a + b;
    c = a + 1;
    // c = 1 + a;
}
```

a.operator+ (1)
1.operator+ (a) ?

The problem with operator member functions is that they are not symmetric. The following works fine because a message is sent to object a with a parameter of the defined type.

c = a + b

The next example also works.

c = a + 1

Once again a message is sent to a to invoke the operator+ method. The problem is that there is no suitable overloaded function for the parameter supplied. Consequently, the compiler looks for a conversion. Since it is possible to construct Fractions from a single int, a direct user-defined conversion is applied.

c = 1 + a

The last example fails. Sending the message operator+ to an integer object with a Fraction parameter is undefined. There is no suitable method in the int class.

Operator Free Functions

```
class Fraction
{
    int numer;
    int denom;
public:
    Fraction (int n=0, int d=1);
    friend Fraction operator+
        (const Fraction& left, const Fraction& right);
};

int main()
{
    Fraction a,b,c;
    c = a + b;
    c = a + 1;
    c = 1 + a;
}
```

operator+ (a,1)

operator+ (1,a)

Using friend functions symmetry is preserved. In both cases where an int argument was supplied the int was coerced into a Fraction, and the corresponding operator called.

Multiple Overloads

```
class Date
{
    int day, mon, year;
public:
    Date (int=1, int=6, int=97);
    Date (const char*);
    friend int operator == (const Date&, const Date&);
    // friend int operator == (const Date&, const char*);
    // friend int operator == (int, const Date&);
    // friend int operator == (const Date&, int);
};

int main()
{
    Date today (15,6,94), raceday;
    if (today == raceday) ...
    if (today == "3/4/63") ...
    if (22 == today) ...
}
```

multiple overloads may be avoided when suitable constructors are available

In some instances it may be necessary to overload an operator for a number of parameter types. Whilst this may solve the problem, the use of constructor conversions is more common. The operator is overloaded only for the class type, and then constructors are provided to enable arguments of different types to be used. This has the tendency to cut down the number of functions which have to be defined.

Where two paths exist for an operation, the most direct path is chosen by the compiler.

Note that relying on such conversions does introduce more overhead since intermediate objects (in the above example Data objects) have to be created and destroyed.

Operator Member Functions

- Some operators can only be overloaded as members

<code>object [arg]</code>	subscript
<code>object (...)</code>	function call
<code>object -> member</code>	pointer
<code>object = arg</code>	assignment

on the side

Overloading any of these operators as friends results in a compilation error.

Take care, the error may be obscure. This restriction does not affect functionality since these operators are asymmetric.

- [] Useful for indexing associative arrays**
- () Simulates a function call, takes multiple args**
- > Unary operator, checks before de-referencing**
- = Implements deep copy**

The subscript operator allows array objects to be implemented. These can be offset from zero (for example, ranging from 10 to 20), can be indexed by any type (so associative arrays indexed by strings could be implemented), and can be designed to provide automatic bounds checking. In fact, although applying the operator to an object may look like an array access, it could be implemented in the object as a file access, a dynamic tree search or perhaps a data base query.

The function call operator allows an arbitrary number of parameters to be passed to the object. There are no implied semantics of operator `()`, unlike all other operators. Its meaning is usually inferred from the name of the object to which it is applied. For example, consider an object of an I/O class:

```
printf ("input value: %d\n", i);
```

The object is called `printf`, and the bracket operator is applied to send a collection of arguments to it. The overall semantics are indicated by the function name, not by the operator itself.

The member selector operator allows smart pointers to be implemented. It is overloaded in a unary form to allow the pointer object to be checked (or setup) before being dereferenced.

The assignment operator allows deep copy (as opposed to shallow copy) to ensure that all members (not just their values) are assigned from one object to the other.

Overloading []

```
class Array {
    char *data;
    int start, length;
public:
    Array (int s, int l) :
        start(s), length (l), data (new char [l])
    {}
    ~Array ()
    { delete [] data; }
    char& operator[] (int i)
    {
        return data[i - start];
    }
};
```

```
int main()
{
    Array a(5,10);

    a[12] = 'g';
    cout << a[12];
}
```

implements an offset array with bounds checking

`Array (start,length)`

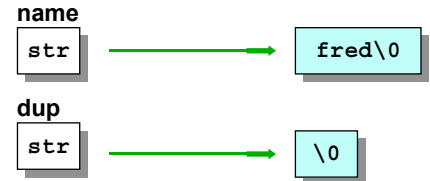
`operator []` is overloaded to provide an index into data held by objects of an array class. The data is actually held as a dynamic array, but could have been held in any form. Typical examples are linked-list structures, data bases and files. The advantage of `operator []` is that it gives access to an object's data in a simple 'array-like' fashion, whilst hiding the way in which the data is physically stored. A reference is returned to the caller so that the operator can be used as an l-value.

`operator []` takes a single argument of any type. In the above the index was an `int`, it could have been `char*`, or `Employee`, or whatever is sensible for the application.

Shallow Assignment

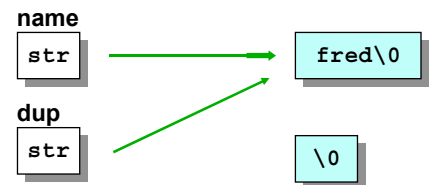
```
class String
{
    char *str;
public:
    String(const char *s = "");
    String (const String&);
    ~String() { delete [] str; }
};

String name ("fred"), dup;
dup = name;
```



The default assignment operator implements shallow copy

This generates a memory fault and a memory leak



The assignment operator should be overloaded for any class which contains pointer members. The default assignment operator, like the default copy constructor, implements shallow copy. Each member is copied, rather than the data it refers to.

Failure to provide an assignment operator is in fact worse than neglecting the copy constructor because a memory leak also occurs.

Overloading =

```
class String
{
    char *str;
public:
    String (const char *s = "");
    String& operator= (const String&);
};

String& String::operator= (const String& r)
{
    if (this != &r)
    {
        delete [] str;
        str = new char [std::strlen (r.str) + 1];
        std::strcpy (str, r.str);
    }
    return *this;
}
```

What is the purpose of this?

Why not return by value?

Why not return by constant reference?

The assignment operator simply copies the information held by one object into another. Where pointers are involved, it must allocate sufficient space for the recipient to store a copy of the data from the source object.

One important detail of the implementation is to avoid self-assignment. For example,

```
a = a;
```

is valid C and C++ code. Without the test in the above program, `str` would be deleted before being copied to itself, resulting in a memory fault.

The return type for assignment is not straightforward. Since the object being assigned-to exists outside of the function, the efficiencies of return-by-reference seem sensible. However, this has an important semantic side-effect. It allows the returned object to be used as an l-value in subsequent expressions:

```
++ (a=10);
```

This behaviour is not usually desirable, so the usual definition is to return a constant reference. Note that for many compilers return-by-value would be just as efficient due to optimisations.

Conversion Operators

- **Constructors are conversion functions**
 - construct user-defined objects from intrinsic objects
 - construct user-defined objects from other user-defined objects
- **Constructors are limited**
 - cannot access intrinsic constructors to add user-defined types
 - cannot anticipate classes not yet invented
- **Operator Conversions**
 - can convert user-defined type to any other type

```
operator int ();  
operator const char *();  
operator Data ();
```

Conversion operators enable classes to define what their objects can convert to. In the above, the operators allow conversions to `int`, `const char*` and `Data`.

Conversion operators are useful because there is no other way of allowing user-defined objects to be coerced into intrinsic objects. They also allow for conversion to user-defined types which do not have suitable constructors.

Conversion Operators

```
class Num
{
    int x;
public:
    Num (int a = 0) : x (a) {}
    operator int () const {
        return x;
    }
};

int main ()
{
    Num a(1), b(2), c;

    a = b + c;
}
```

Does this compile?

With conversion operators and constructors compilers are able to construct objects of one type from another implicitly. In order to control the extent of the coercions, at any stage in an evaluation only one user-defined and one intrinsic conversion may take place.

In the above program, the `Num` objects are converted to `int`'s, added, and then cast to a `Num` object for assignment.

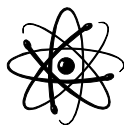


Chapter 15: Single Inheritance

- **Single Inheritance**
 - **Class Derivation**
 - **Contents of Derived Class**
 - **Access Regions**
 - **Protected Access**
 - **Object Construction**
 - **Private Inheritance**
 - **Resolving Scope Conflicts**
 - **Propagating to Base Class Methods**
-

Single Inheritance

- **A new class inherits members from an existing class**
- **All the base class members appear in the child**
 - child may add new members
 - override inherited members
 - hide inherited members
- **Inheritance is key to object-oriented programming**
 - supports 'programmable re-use'
 - underpins polymorphism
 - enables specialisation



Object oriented languages are built around the concept of inheritance. Inheritance is the class relationship of 'Is A' or 'Kind of', and is used when one class is a specialisation or refinement of another.

Single inheritance is where a class derives all the members (both state and functions) from a single parent (or base) class. The child can subsequently override the parents behaviour (by providing its own members with the same identifier), hide inherited members (by making them inaccessible to objects), and add new members.

Inheritance strongly supports the idea of re-use. It is founded on the concept that everyday and abstract things can be classified. As such, certain common characteristics can be extracted, generalised, and placed high in the inheritance hierarchy to use over again.

Re-use is not just concerned with saving development costs, but also in saving (the more significant problem) maintenance and modification costs. Classes which are commonly re-used, are mature, tried and tested.

Class Derivation

```
#include <string>

class Person
{
    std::string name;
    int age;
public:
    Person (std::string, int);
    void display() const;
};

class Student : public Person
{
    std::string course;
    Date start;
public:
    Student (std::string, int, const Date&);
    void join_course (std::string);
    void display() const;
};
```

← inherit Person members

← add new members

← override inherited member

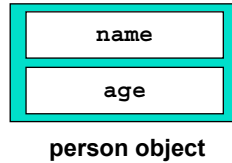
Student is a kind of **Person** with two additional data members and one extra member function. Moreover, **Student** overrides the parent's **display** method with its own.

Note that at this time the parent's **display** function is still available to objects of **Student**. However, the intention is usually that **Student's display** should output the local members, and then pass control to **Person's display**.

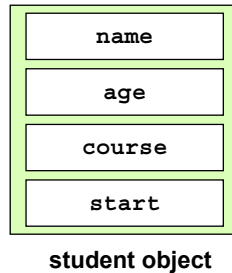


When overriding inherited functions the interface of the child function must be identical to that of the parent. (Strictly speaking, the return type can be less restrictive and the input parameters may be more restrictive.)

Memory Layout of Objects



`Person::display()`



`Person::display()`
`Student::display()`
`Student::join_course()`

The memory layout of `Student` includes all data members of the parent class `Person`. The member functions for `Student` include all of those defined for `Person`.

Note that not all member functions and data members inherited from a parent are accessible to either the derived class or its objects. Member visibility depends on member access rights and the type of inheritance employed.

Access Regions

- **Private**
 - visible only to member functions and friends

- **Protected**
 - visible to member functions and friends
 - visible to member functions of derived classes

- **Public**
 - visible to member functions and friends
 - visible to member functions of derived classes
 - visible to objects

Private and public access are as previously stated. They respectively give no access or complete access to the class members.

Protected access allows members of the base class to be visible to the derived class, but not to objects of the class. This allows a class designer to keep the public interface simple (minimising coupling in the user program), whilst giving greater control to a derived class to refine the behaviour.

Protected Access

```
class A
{
    int x;
protected:
    int y;
public:
    int z;
};

class B : public A
{
    void update () const
    {
        // ++x;
        ++y;
        ++z;
    }
};
```

```
int main()
{
    B fred;

    // ++fred.x;
    // ++fred.y;
    ++fred.z;
}
```

← attempted access to private member



Access to private base class members is prohibited in the derived class; access to both private and protected members is prohibited to objects.

Although protected members are not visible to objects, they do increase the coupling between base and derived classes. They should therefore be minimised, or if this is not possible, limited to protected member functions.

Object Construction

- **Base constructs base, derived constructs derived**

```
Derived::Derived (arguments) : Base (arguments) {}
```

- **Separates responsibilities**
 - derived class should not be required to re-write base code
 - derived cannot touch private base members
- **Construction order**
 - order of declaration (not order of initialisation list)
 - inherited classes
 - composed objects
 - local construction

The purpose of deriving one class from another is to promote re-use. In addition to inheriting the basic members of the base class, the child also inherits the constructor and destructor functions of the base.

Constructor and destructor functions for both the base and derived class are called implicitly when objects of the derived class are instantiated. Once the memory is allocated, control is passed first to the base class constructor, and then to the derived constructor. Destruction occurs in the reverse order.

To enable the derived class to pass constructor values to the base class the initialisation list can take base class type specifiers (plus arguments).

Object Construction

```
#include <string>

class Person
{
    std::string name;
    int age;
public:
    Person (std::string, int);
    void display() const;
};

class Student : public Person
{
    std::string course;
    Date start;
public:
    Student (std::string, int, const Date&);
    void join_course (std::string);
    void display() const;
};

Student::Student (std::string n, int a, const Date& d)
: Person (n, a), start(d)
{}

int main()
{
    string dude ("fred");
    Date begin (10,09,97);

    Student s1 (dude, 29, begin);
}
```

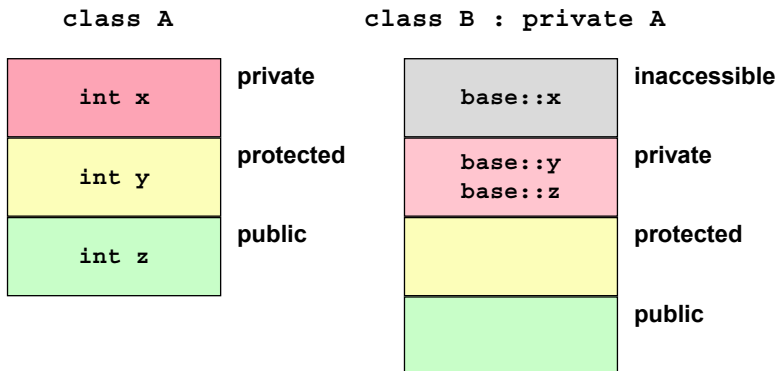
In order to make an instance of `Student`, the `Student` class must provide constructor values for its base class. The only exception is where the base class has a default constructor.

The `Student` constructor receives three arguments which are subsequently used to initialise first the base class `Person`, and then the composite member `start`. The other `String` component of `Student` is not initialised explicitly (it must therefore have a default constructor).

`display` in the derived overrides `display` in the parent. Recall that `display` is `const` because it does not affect the state of the object to which it is applied. Note also that where possible `const string&` have been used to pass arguments. The `&` is for pointer efficiency and the `const` to indicate that the parameter will not be changed.

Private Inheritance

- All members of the base class are private to objects of the derived class



So far only public inheritance has been discussed. There are in fact two other flavours of inheritance available. Private inheritance is used to hide all of the member of the base class from both objects of the derived and from future derivations. In private inheritance a wrapper is placed around all members of the base such that in order to access these members specific interface functions must be provided.

Protected inheritance is a somewhat less constraining form of derivation. In this simply the public members of the base are moved into the protected region of the derived. The purpose is to hide all base class members from objects, but to keep such members visible to subsequent derivations.

		Type of Inheritance		
		private	protected	public
access region in base	private	X	X	X
	protected	private	protected	protected
	public	private	protected	public
		Resulting access in derived		

Private Inheritance

```
class Bird {
    // data members
public:
    void fly (int altitude, int speed);
    void eat (String& food);
    void sleep (const Time& hours);
};

class Ostrich : private Bird {
public:
    void eat (String& f) {
        Bird::eat (f);
    }
    void sleep (const Time& h) {
        Bird::sleep (h);
    }
};
```

**An Ostrich can't fly,
but can do everything
else a Bird can do.**

Provides implementation inheritance with no interface inheritance. Composition is often used instead.

Private inheritance allows all of the members of the base class to be inherited into the derived without being visible in the child's public interface. This is pure implementation inheritance.

In the above example an *Ostrich* is only a kind of *Bird* because the fly method for *Bird* is hidden. The other methods, *eat* and *sleep*, are propagated to the base class explicitly. This is necessary because none of the base class methods are visible in the interface of *Ostrich*.

Design Note

Private inheritance is not often used because composition is usually equivalent and is somewhat simpler. Also, from a modelling perspective it is not true to say that an *Ostrich* is a kind-of *Bird*, because an *Ostrich* object cannot be used in every context in which a *Bird* is expected.

Private inheritance is sometimes used when virtual functions are defined in the Base class and the derived class wishes to take advantage of them. However, even in this case public inheritance is used more often. The issues is discussed in the polymorphism chapter.

Resolving Scoping Conflicts

```
int x;

class Base
{
public:
    int x;
};

class Derived : public Base
{
public:
    int x;
    void update (int x) const
    {
        ++x;
        ++Derived::x;
        ++Base::x;
        ++ ::x;
    }
};
```

**Use the scope operator
to resolve scope conflicts.**

```
int main()
{
    Derived obj;

    ++obj.x;
    ++obj.Base::x;
}
```

The only scoping problems which result from single inheritance are when a local identifier is used in place of the one expected. The scope resolution operator can resolve most of these problems. However, some consideration to naming schemes should also be given.

Propagating to Base Class Methods

```
class Parent
{
    int data;
public:
    Parent (int d) : data (d) {}
    Parent (const Parent& right) {
        data = right.data;
    }
    Parent& operator= (const Parent& right) {
        data = right.data;
        return *this;
    }
    void display() const {
        std::cout << data << std::endl;
    }
};

class Child : public Parent
{
    int stuff;
public:
    Child (int d, int s);
    Child (const Child& right);
    Child& operator= (const Child& right);
    void display() const;
};
```

The above code fragment shows a simple base class with a copy constructor and `operator=` overloaded. In practice, these methods are only necessary when the class contains pointer members. Since this example class does not contain pointers the implementations are much simpler.

In the `Child` class a copy constructor is also provided along with an overloaded assignment operator. The question is, how are these implemented? What should the copy constructor taking a `Child&` argument propagate to the base class copy constructor? Similarly, what should the assignment operator pass on to the base class operator? Also, how can the latter even be called?

Propagating to Base Class Methods

```
#include <iostream>
#include "Child.h"

Child::Child (int d, int s)
: Parent (d), stuff (s)
{}

Child::Child (const Child& right)
: Parent (right)
{
    stuff = right.stuff;
}

Child& Child::operator= (const Child& right)
{
    Parent::operator= (right);
    stuff = right.stuff;
    return *this;
}

void Child::display() const
{
    Parent::display();
    std::cout << stuff << std::endl;
}
```

**right is coerced
into a Parent**

**operator= is called
explicitly & right is
coerced to Parent**

Both the copy constructor and overloaded assignment simply pass the Child object parameter, `right`, straight through to the corresponding base class method. Since a Child object IS-A kind of Parent object, the derived may be naturally (implicitly) coerced to the base. This works because base objects have a subset of the members contained in derived objects. For the same reason, the reverse conversion does not work.

In order to scope `operator=` to the Parent class, it is called using the functional name rather than with the operator symbol.

Course Notes



Chapter 16: Polymorphism

- The Nature of Inheritance
 - Conversions between Base & Derived Classes
 - Static Binding
 - Dynamic Binding
 - Virtual Functions
 - Polymorphism
 - Pure Virtual Functions
 - Run-Time Type Information
 - Virtual Destructors
-

The Nature of Inheritance

```
#ifndef PERSON_H
#define PERSON_H

#include <string>

class Person
{
    std::string name;
public:
    Person (const std::string&);
    void display() const;
};

class Student : public Person
{
    std::string course;
public:
    Student (const std::string&, const std::string&);
    void display() const;
};

#endif
```

Student is a kind-of Person, and has all of the members of Person, in addition to some of its own.

During inheritance a new class is derived from an existing class. The derived (child) class possesses all of the state and behaviour of the base (parent) class, in addition to some of its own. The derived class can add new members, override inherited members or hide inherited members by changing their access region.

It is the nature of inheritance, therefore, that all derived classes are 'kinds-of' their base classes. They all contain the base class members and could therefore be used in a context in which objects of the base class are expected.

This is known as Liskov's principle of substitutability and contravariance.



As a general rule, it is better to avoid including header files within header files since such inclusion increases the physical coupling in the program. In the example code above, `string.h` and `date.h` are included because instances of these objects are used in the `Person` and `Student` classes. In order for the compiler to even size a `Person` object it must see the definition (not merely a forward declaration) of the `String` object from which `Person` is composed.

The Nature of Inheritance

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

Person::Person (const std::string& n)
    : name (n)
{}

void Person::display () const {
    std::cout << "person: " << name << std::endl;
}

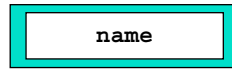
Student::Student (const std::string& n, const std::string& c)
    : Person (n), course (c)
{}

void Student::display () const {
    std::cout << "student: " << course << " / ";
    Person::display();
}
```

For completeness the members of the `Person/Student` hierarchy are shown above. Note that `Student::display` propagates the request to `Person::display` so that the private members of `Person` are also printed.

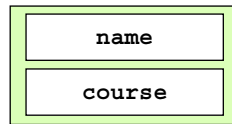
Note that the compiler provided copy constructors and overloaded assignment operators is sufficient for the `Person` and `Student` classes.

Conversions Between Base & Derived Classes



person object

There is no direct conversion from base class to derived because the derived requires extra members



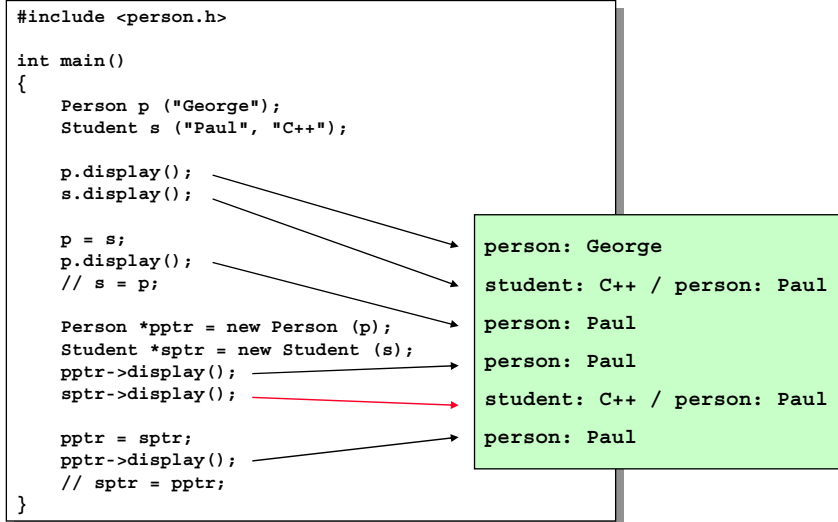
student object

There is a natural conversion from derived to base, because all of the base members are in the derived class

The only way to convert from a base class to a derived class is to provide a suitable constructor. This is not required, however, when converting the other way.

Since all derived classes contain their base class members their objects can be used as base class objects. There is no need to provide extra constructors or conversion operators. The relationship is considered to be a 'natural conversion' in the language.

Conversions Between Base & Derived Classes



In the above example various objects of the base class and derived class are created. Attempts to coerce derived class objects into base class objects are successful, attempts to coerce base class objects into derived fail.

The reason why it is possible to assign the `Student` object `s` to the `Person` object `p` is that `s` is a superset of `p`. `s` contains all of the members expected for an object of type `p`. Consequently, the compiler only accesses those members of `s` which are also in `p`; it treats `s` as an object of `Person`. The attempt to assign the `Person` object `p` to the `Student` object `s` fails because `p` cannot be viewed as a `Student`. It does not have all of the members of a `Student`, and the compiler does not know how to create those missing. (A suitable conversion operator or constructor would have to be created for this to work.)

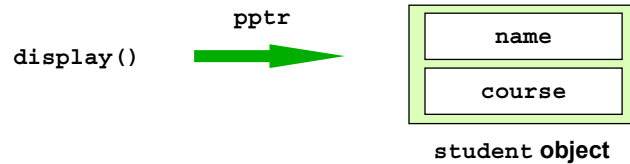
The natural conversions also apply when objects are referenced through pointers. It is possible to assign the address of a derived object to a base class pointer, but not the converse. When the derived object is accessed through the base class pointer, it behaves as a base class object. In the above, for example, when the `display` message is sent to the `Student` object through the base class pointer, `pptr`, the base class `display` method is called.

```
pptr -> display();
```

In object oriented terms this behaviour is wrong. The `display` message is sent to a `Student` object. According to the rules of encapsulation, the object should decide how it is to be displayed, and therefore `Student::display()` should have been called.

Static Binding

- Compile time static binding restricts OO behaviour



- Behaviour based on pointer type not object type
 - Intrinsically not object-oriented
 - behaviour should be devolved into objects

The message `display` is sent to a `Student` object through the base class pointer, `pptr`.

In static binding (also known as early or compile time binding) the compiler is not concerned with what `pptr` points at. Indeed, it cannot be interested since this will change during the execution of the program. The compiler is only concerned with `pptr`'s type. Thus

```
pptr -> display()
```

is interpreted as

```
Person::display (pptr)
```

`Person::display` only knows about those parts of `Student` which are also parts of `Person`, and therefore accesses the `Student` object as a `Person`.

Object-Oriented Behaviour

In object oriented terms `pptr` is simply a means of talking to the object, it is a path through which messages travel. Ultimately the `display` message is delivered to an object of class `Student`, and as a consequence the `Student::display` method should be called. In this arrangement the type of the pointer is irrelevant.

This behaviour is highly desirable. It allows objects to be self managing, and decouples the logic of a program from the objects which it manipulates.

Dynamic Binding

- **Selecting the appropriate behaviour at run-time**
 - dynamically binding to object's method
- **The choice is based on object type not pointer type**
 - when an object receives a message to display, it decides how this should be processed; not the messenger!
- **The mechanism facilitates polymorphism**
 - functions defined for base class objects may be applied to any derived class object without loss of state or behaviour
- **Dynamic binding is enabled through virtual functions**
 - the keyword virtual is used to qualify base class pointers
 - causes a function table pointer to be associated

In C++ member function selection can be deferred until run-time. As such it can be based on what is pointed to rather than the type of pointer being used. This behaviour is known as dynamic (or sometimes late or run-time) binding.

Dynamic binding is the norm in object-oriented systems. Messages are sent to objects, and depending on the object being addressed (as opposed to the type of pointer) a function appropriate to the object is invoked.

In C++ dynamic binding is established through inheritance and virtual functions. A function is declared virtual in the base class and then overridden in the derived. When invoked through a base class pointer, the derived function is actually called. Since there are likely to be many derived classes from a single base, the selection of the function to be called is established dynamically at run-time. Virtual functions tend to have many implementations (one appropriate for each derived class) and are consequently referred to as polymorphic.

Virtual Functions

```
#ifndef PERSON_H
#define PERSON_H

#include <string>

class Person
{
    std::string name;
public:
    Person (const std::string&);
    virtual void display() const;
};

class Student : public Person
{
    std::string course;
public:
    Student (const std::string&, const std::string&);
    virtual void display() const;
};

#endif
```

virtual in the derived
is implicit and does not
need to be restated

To enable dynamic binding, functions in the base class must be qualified with the word 'virtual'. This causes a function table to be associated with the class, and a pointer to the table to be included with each object of the class.

In derived classes the child inherits the parent's members and may override the implementations. Virtual functions in the base class remain virtual in the derived. Each of the derived class objects contains a pointer to the classes' virtual function table. This gives each object knowledge of its methods, thus removing the need to select the methods based on pointer type.



Note that the prototypes of overloaded functions in the derived class must be consistent with those of the base. If this is not the case, the derived functions 'hide' the base class functions and dynamic binding cannot occur.

Virtual Functions

```
#include <person.h>

int main()
{
    Person *pptr = new Student ("Paul", "C++");
    pptr->display();
}
```

```
$ ./a.out
student: C++ / person: Paul
```

The pointer type no longer determines behaviour

The correct display function is determined at run-time due to the type of the object referenced.

In the above the message `display` is sent to the object pointer to by `pptr`. Since the object is of type `Student`, and the `display` method is declared virtual in the base class `Person`, dynamic binding ensures that `Student::display()` is called.

Polymorphism Example

```
class Animal {
public:
    virtual void move() {
        std::cout << "generic movement" << std::endl;
    }
};

class Frog: public Animal {
public:
    void move() { std::cout << "frogs hop" << std::endl; }
};

class Bird: public Animal {
public:
    void move() { std::cout << "birds fly" << std::endl; }
};

class Snake: public Animal {
public:
    void move() { std::cout << "snakes slither" << std::endl; }
};

class Horse: public Animal {
public:
    void move() { std::cout << "horses trot" << std::endl; }
};
```

In this example the base class `Animal` is derived into a series of child classes. The generic behaviour for animal objects is to 'move'. The method is introduced as virtual in the base, and then overridden by each of the derived so that the behaviour is appropriate for the kind-of animal being defined.

Since `move` is defined for a number of distinct classes, having the same semantics but different implementations for each, it is a polymorphic function.

Note that dynamic binding only works with public inheritance. If private or protected inheritance is used, then the virtual function is inaccessible to both derived objects and pointers to derived objects.



Polymorphism Example

```

void doit (Animal *p)
{
    p->move();
}

int main()
{
    Frog f;
    Bird b;
    Snake s;
    Horse h;

    doit (&f);
    doit (&b);
    doit (&s);
    doit (&h);

    Animal *array[] =
        { &f, &b, &s, &h };
    for (int i=0; i<4; i++)
        array[i]->move();
}

```

C pseudo code
Note explicit case analysis

```

void doit (void *p, int type)
{
    switch (type) {
        case 'h': (Horse *)p -> move();
                    break;
        case 'b': (Bird *)p -> move();
                    break;
        ...
    }
}

```

In the above example a variety of 'kinds-of' animal object are created. Their addresses are stored in an array of base class pointers. Through the base class pointers, the move message is sent to the objects. Due to dynamic binding, each animal moves in an appropriate fashion due to its class, not to the type of the pointer.

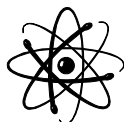
In the `doit()` function an animal object is supplied, and the function issues the message move. The particular 'kind-of' animal is not relevant, since each deal with their own implementation of move.

The Power of Polymorphism

The advantage of polymorphism and dynamic binding is that the program may be extended, in this case with more animals, with no or little effects on the code. The `doit()` function does not have to be extended to cope with other kinds of animal, not even for those not yet thought of.

By contrast, the C code requires information relating to the type, so that the compiler can bind the pointer to the correct function. Such case statements are not uncommon in C, and lock the application to the set of objects for which it was originally designed. The problem with case statements is that they

- are often large and tedious to write
- are slow to execute as many conditions may have to be tested
- may be duplicated throughout the program
- make the program inflexible since all options are "hard coded"
- may omit cases



Polymorphic functions (such as move above) decouple the program logic from the entities which are being managed. This encourages re-use, and makes the software more amenable to change.

Pure Virtual Functions

- **Class Animal established a common interface**
 - no need to instantiate Animal in this system

```
class Animal
{
public:
    virtual void move() {
        std::cout << "generic movement" << std::endl;
    }
};
```

- **Pure virtual functions only need to be declared**

```
class Animal
{
public:
    virtual void move() = 0;
};
```

The class cannot be instantiated, and is therefore abstract

C++ implements polymorphism and dynamic binding through inheritance. The base class must introduce virtual methods which are subsequently overridden in the child classes.

A problem of this mechanism is that it requires the programmer to provide definitions of base class virtual functions even though it may not be desirable (or meaningful within the program) to make objects of the base class.

To resolve the problem C++ allows virtual functions to be declared as pure. This is achieved by initialising the function to 0. It is subsequently not necessary to provide a definition in the base class. Moreover, since the base class is no longer fully defined, it is not possible to make objects of this class. The base class is referred to as an abstract class.

Child classes of an abstract base class are required to provide definitions for all pure virtual functions. If they fail to do so, they too are abstract and uninstantiable.

Run-Time Type Information

- Sometimes necessary to determine concrete type
 - special cases may occur and need to be safe

```
class Bird : public Animal
{
public:
    void fly () {
        ...
    }
};
```

Birds can fly but not
all Animals can

- If we think we have a bird, we can try to fly ...

```
void doit (Animal *p)
{
    Bird *b = static_cast<Bird *>(p);
    b->fly();
}
```

but if *p* is not pointing to
a bird, we'll crash land!

Run-Time Type Information

- **C++ support RTTI for type-safe down (cross) casts**
 - used in polymorphic (virtual) inheritance hierarchy

```
void doit (Animal *p)
{
    if (Bird *b = dynamic_cast <Bird *>(p))
        b->fly();
}
```

- **dynamic_cast operator attempts to cast the object**
 - returns address if *p* really refers to a bird
 - returns 0 otherwise

The `dynamic_cast` is part of the RTTI mechanism in C++. It allows the type of an object to be inspected at run-time, so that safe 'down casting' within inheritance hierarchies can be made.

The `dynamic_cast` operator takes two arguments, the type required and a pointer to cast. The operator inspects the type of the objects being cast, and if the conversion is correct returns a pointer to the object suitably cast. If, however, the object is not of an appropriate type then the operator returns 0.

The RTTI mechanism is built by extending the virtual function table (vtbl) mechanism which C++ provides to support inclusion (inheritance) polymorphism. Essentially, the virtual function table associated with each class is associated with an object of a `type_info` class, which provides information to allow objects of the class (carrying a pointer to the vtbl) to be identified at run-time. Note that the whole mechanism assumes the existence of a vtbl, so will only work when virtual functions are introduced in the base class.

Extended Type Information

- RTTI based on `type_info` language class

- methods to determine class name
- some basic operations such as `==` and `!=`

```
#include <iostream>
#include <typeinfo>

class Data
{
    int i;
public:
    Data (int j = 0) : i (j) {}
    virtual ~Data () {}
};

int main()
{
    Data *p = new Data (10);
    cout << typeid (*p).name() << endl;
}
```

Use these mechanism sparingly; ideal is to be polymorphic!

In the example code on the left, the `name()` method is used to determine a string representation of the type name. The `type_info` object is accessed implicitly by the `dynamic_cast` operator. It is also accessed using the comparison operators as shown on the following page.

The RTTI mechanisms should be used sparingly because they bring type dependent behaviour explicitly into the code. The problem with explicit type analysis (or case analysis using a switch statement) is that it tends to hardwire programs to the specific set of object types envisioned when the program was written---and makes it difficult to evolve the program should the set change. In this sense, overuse of the RTTI mechanisms undermines polymorphism. However, there are some cases when the ability to determine the type of an object is necessary and may even improve the overall structure and type safety of a program. For example, if all but one class in a hierarchy conform to a polymorphic interface, and the exception has an additional method. A function acting on derived objects through a base pointer, and intending to call the extra method, should use the `dynamic_cast` to ensure that the object is of the special type.

Virtual Destructors

```
class Animal
{
public:
    virtual void move() = 0;
    ...
    virtual ~Animal();
};

class Bird : public Animal
{
public:
    void fly ();
    ...
}

int main()
{
    Animal *p = new Bird (...);

    p->move();
    delete p;
}
```

← Destructor must be virtual otherwise only base class destructor called.

← Bird's destructor called and then Animal destructor



Whenever dynamic binding is being used, it is wise to also make the base class destructor virtual. The consequence, otherwise, is that derived objects deleted through base class pointers will not be properly destroyed. The compiler will statically bind the base class destructor to the pointer, and the derived class destructor will not be called.

Chapter 17: Multiple Inheritance

- Multiple Inheritance
 - Initialising Base Classes
 - Thermostat Example
 - Resolving Scope Conflicts
 - Duplication of Base Class Members
 - Virtual Base Classes
 - Resolving Constructor Conflicts
-

Multiple Inheritance

- Multiple inheritance allows new classes to be derived from multiple base classes
- Base classes are usually distinct, derived class exhibits equally the behaviour of all bases
- Objects of the resulting class may behave as objects of any of the base classes

The original reason for the introduction of multiple inheritance was to allow two or more classes to be combined into one, so that objects of the resulting class would behave as objects of either base class.

This allows objects which naturally belong to two or more classes to be manipulated by parts of the program oriented towards each of their bases. The advantage is that the functionality does not have to be built explicitly into one class library (making it bloated) or through tight inter-class coupling. Multiple inheritance provides a means of organising libraries around simpler classes with fewer inter-class dependencies.

Multiple inheritance is a precise way of modelling the natural relationship of an object exhibiting the characteristics of two or more things.

A sales manager

Given a `Manager` class and a `SalesPerson` class, multiple inheritance would allow their natural amalgamation into a `SalesManager` class. Members of both base classes appear (at the same semantic scope) within the `SalesManager` derived class.

Multiple Inheritance

```
class BaseOne {  
    // members  
};  
  
class BaseTwo {  
    // members  
};  
  
class Derived : public BaseOne, public BaseTwo {  
    // additional members  
};
```

- **Single and multiple inheritance are similar**
 - private, public and protected
 - derived class passes values to bases via initialisation list
 - scope operator resolves identifier ambiguities
 - construction follows order of declaration in derivation list

Multiple inheritance behaves exactly as single inheritance. Members of the base class appear in the derived, but may be moved into a different access region or overridden.

The derived class above is a 'kind-of' `BaseOne` and a kind of `BaseTwo`. The order of the derivation list (`BaseOne` then `BaseTwo`) dictates the order of initialisation. As with single inheritance, the base classes are responsible for constructing themselves, and the derived for constructing itself. The derived class passes values to the base class constructors via its initialisation list.

Where ambiguities occur between base class members with the same identifier, the scope resolution operator should be used. There are no ambiguities between the derived class members and the members of the base classes since the former is in the most immediate scope.

Thermostat Example

```
#ifndef GAUGE_H
#define GAUGE_H

class TempGauge
{
    double temp;
public:
    TempGauge ();
    void setTemp (double);
    double getTemp ();
};

class Switch
{
    enum {OFF, ON};
    int state;
public:
    Switch ();
    void toggle ();
    int isOn ();
};

#endif
```

```
#ifndef THERMOSTAT_H
#define THERMOSTAT_H

#include <gauge.h>

class Thermostat:
    public TempGauge, public Switch
{
    double reqTemp;
public:
    Thermostat (int = 0);
    void setReqTemp (double);
    double getReqTemp ();
    void setTemp (double);
};

#endif
```

**A Thermostat is a kind-of
TempGauge and a kind-of Switch**

In this example a `Thermostat` is modelled as a kind-of temperature gauge and a kind-of switch. The `TempGauge` records the current temperature and has two methods. `setTemp` informs `TempGauge` of a new temperature, while `getTemp` queries the current temperature. The `Switch` class has two states, `ON` and `OFF` and provides two methods. `toggle` flips the switch and `isOn` is true when the switch is in the `ON` state.

`Thermostat` maintains the required temperature in `reqTemp` along with methods to alter this temperature. It also overrides the `setTemp` method inherited from `TempGauge`. When `setTemp` (indicating a change in temperature) is called on a `Thermostat`, then a heater or cooler may be switched on or off, in addition to the temperature being recorded.

Thermostat Implementation

```
#include <gauge.h>

TempGauge::TempGauge () : temp (0) {}
void TempGauge::setTemp (double t) { temp = t; }
double TempGauge::getTemp () { return temp; }

Switch::Switch () : state (OFF) {}
void Switch::toggle() { state = state ? OFF : ON; }
int Switch::isOn () { return state; }
```

```
#include <thermostat.h>

Thermostat::Thermostat (int r) : reqTemp (r) {}
void Thermostat::setReqTemp (double r) { reqTemp = r; }
double Thermostat::getReqTemp () { return reqTemp; }

void Thermostat::setTemp(double t) {
    TempGauge::setTemp (t);
    if ((isOn() && getTemp() > reqTemp) ||
        (!isOn() && getTemp() < reqTemp))
        toggle();
}
```

The **Thermostat** uses its **Switch** to control a heater, and its **TempGauge** to monitor the current temperature.

Each time the **Thermostat** receives a **setTemp** event it updates the current temperature and enables or disables the heater accordingly.

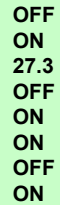
Using the Thermostat

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

int main()
{
    Switch s;
    std::cout << (s.isOn() ? "ON" : "OFF") << std::endl;
    s.toggle();
    std::cout << (s.isOn() ? "ON" : "OFF") << std::endl;

    TempGauge g;
    g.setTemp (27.3);
    std::cout << g.getTemp() << std::endl;

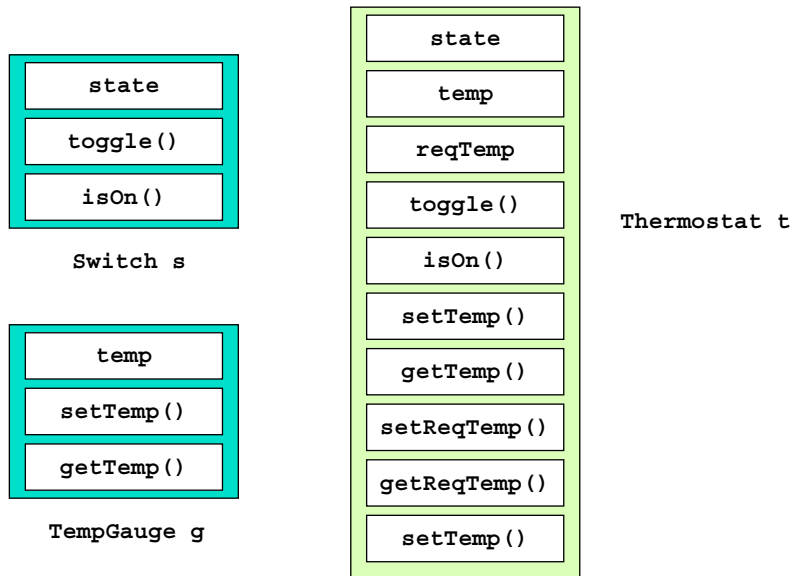
    Thermostat t(30);
    std::cout << (t.isOn() ? "ON" : "OFF") << std::endl;
    t.setTemp (24.5);
    std::cout << (t.isOn() ? "ON" : "OFF") << std::endl;
    t.setTemp (28.5);
    std::cout << (t.isOn() ? "ON" : "OFF") << std::endl;
    t.setTemp (30.1);
    std::cout << (t.isOn() ? "ON" : "OFF") << std::endl;
    t.setTemp (29.8);
    std::cout << (t.isOn() ? "ON" : "OFF") << std::endl;
}
```



OFF
ON
27.3
OFF
ON
OFF
ON

The `Thermostat` object `t` is initially **OFF** until it receives a message about the current temperature. It then switches on until the temperature reaches the desired value.

Multiple Inheritance



The data and function members of the three classes in the thermostat example are shown above. Objects of class `Thermostat` contain all of the members of `Switch` and `TempGauge`, and the additional members introduced in this class.

Scoping Conflicts

```
class Animal {
public:
    std::string name;
    int code;
};

class Disease {
public:
    std::string name;
    int code;
};

class BadPet : public Animal, public Disease {
    int code;
public:
    void display () const {
        std::cout << code;
        std::cout << name;
    }
};
```

**A scope conflict exists for the identifier name.
There is no conflict for code.**

The above, somewhat sad example, models diseased animals as kinds of Animal and kinds of Disease. This is an instance where composition seems more appropriate than multiple inheritance.

The above program will not compile. There is an ambiguity in BadPet between Animal's name and Disease's name. When name is used in BadPet it is not clear to the compiler whose name should be used. (Note that both exist in the derived class.)

There is no conflict with the code member, however, because this is redefined in the derived class. Since this definition is in the most immediate scope, BadPet's code is used.

Resolving Scope Problems

```
class Animal {
public:
    std::string name;
    int code;
};

class Disease {
public:
    std::string name;
    int code;
};

class BadPet : public Animal, public Disease {
    int code;
public:
    void display() const {
        std::cout << Animal::code << std::endl;
        std::cout << Disease::code << std::endl;
        std::cout << Animal::name << std::endl;
        std::cout << Disease::name << std::endl;
    }
};
```

Use the scope resolution operator to solve scoping conflicts

The scope conflicts are easily resolved using the scope resolution operator.

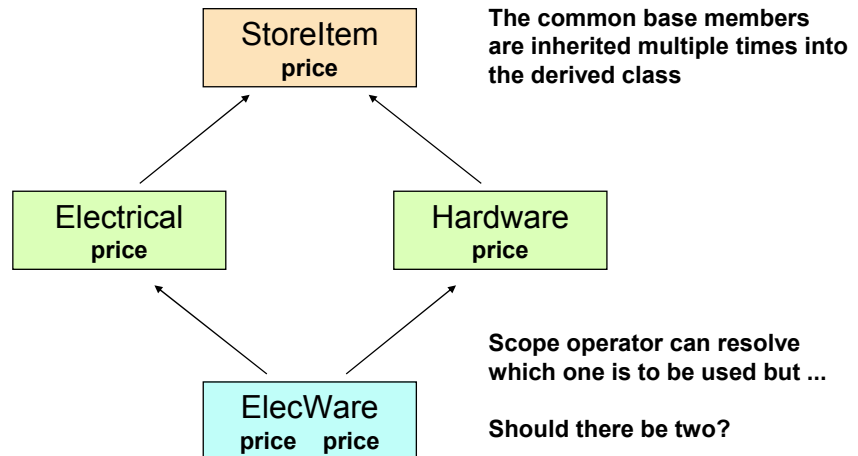
Note that members with the same identifiers in the public interfaces of the base classes (and also the derived) can be accessed through objects using the scoping operator.

For example, if in the above the `code` member was in the public interface of each of the classes, then the three variables could be accessed by objects directly.

```
BadPet dog;

dog.code;           // BadPet's code
dog.Animal::code;   // Animal's code
dog.Disease::code;   // Disease's code
```

Duplication of Base Classes



Multiple inheritance has not been accepted by all parts of the C++ community. One problem which it introduces occurs when members of some common base class are repeatedly inherited into the ultimate derived class, when this is not wanted.

For example, in a store program a collection of classes have been developed to facilitate stock control. The `StoreItem` class models behaviour common to all products, namely a `price`. Using single inheritance this is specialised into `Electrical` items and `Hardware` items. Thus the `price` member is available now in both `Electrical` and `Hardware`. Multiple inheritance is then used to model products which are both electrical and classified as `Hardware`. The `ElecWare` class has all the members of `Electrical` and all the members of `Hardware`. Since `price` appears in both of these, it now appears twice in `ElecWare`.

In some situations it may be desirable for members of a common base to appear repeatedly in a derived class. For example, a `name` in the `Animal` class and a `name` in the `Disease` class may be used to hold different information. However, sometimes (and in the case of the store example) the base members should appear only once. `ElecWare` items should only have one `price` not two.

Virtual Base Classes

- Ensures that members of a common base occur once

```
class StoreItem
{
    int price;
};

class Electrical : virtual public StoreItem
{
    // members
};

class Hardware : virtual public StoreItem
{
    // members
};

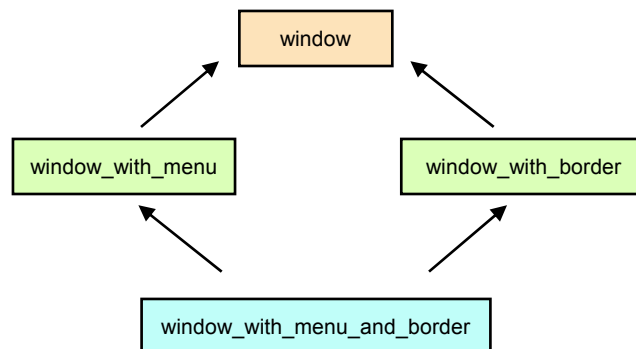
class ElecWare : public Electrical, public Hardware
{
    // members
};
```

virtual causes members of the base class to appear once in the multiply-inherited child class

C++ provides a facility known as virtual inheritance to avoid duplication of base class members in ultimately derived classes.

Both `Electrical` and `Hardware` are virtual `StoreItem`'s. This means that should they be subsequently involved in a multiple inheritance, their common base (`StoreItem`) will occur only once.

Multiple-inheritance with virtual base classes is used in several window class libraries to model the relationship between components



Note, if `ElecWare` inherited from a third kind-of `StoreItem` which was not derived using virtual inheritance, then its `StoreItem` component would be repeated within `ElecWare`.

Constructor Conflicts

```
class StoreItem
{
    int price;
public:
    StoreItem (int i) : price(i) {}
};

class Electrical : virtual public StoreItem
{
public:
    Electrical (int i) : StoreItem(i) {}
};

class Hardware : virtual public StoreItem
{
public:
    Hardware (int i) : StoreItem (i) {}
};

class ElecWare : public Electrical, public Hardware
{
public:
    ElecWare (int i) : Electrical (i), Hardware (i) {}
    ElecWare () : Electrical (5), Hardware (10) {}
};
```

This program will not compile

There are potentially two initialisation paths for the common base member price.

This program will not compile because the common base members (which now occur just once because of virtual inheritance) are initialised twice. The problem with this is that the initialisation values may not be the same in each case.

In the above example, the constructor for ElecWare passes on constructor requests to both the Electrical and Hardware base classes. These in turn pass on constructor requests to their respective StoreItem component. In ElecWare members of the StoreItem base class appear only once, but are initialised through both the Electrical base and the Hardware base.

In the default constructor above, it is not clear whether the price from StoreItem is set to 5 or 10.

Resolving Constructor Conflicts

```

class StoreItem
{
    int price;
public:
    StoreItem (int i) : price(i) {}
    StoreItem () : price (0) {}
};

class Electrical : virtual public StoreItem
{
public:
    Electrical (int i) : StoreItem(i) {}
};

class Hardware : virtual public StoreItem
{
public:
    Hardware (int i) : StoreItem (i) {}
};

class ElecWare : public Electrical, public Hardware
{
public:
    ElecWare (int i) : Electrical (i), Hardware (i), StoreItem(i) {}
    ElecWare () : Electrical (5), Hardware (10) {}
};

```

The void constructor is called in the event of ambiguity

The StoreItem component is explicitly initialised

The constructor conflict is resolved in one of two ways. Either the ultimate base class provides a default (void) constructor to be called whenever an ambiguity occurs; or the ultimate derived class explicitly initialises the common virtual base component.

Neither of these solutions are good. Using a void constructor when arguments are actually supplied may be misleading to programmers. Values are being provided and subsequently ignored.

The second scenario of initialising the base component explicitly exposes too much information. Why should the ultimate child class have to know about the ancestry of its parents?

These problems have caused many people to suggest that multiple inheritance should not be supported in the language. However, this situation only arises when there is a common base class, and often therefore one designer. Multiple inheritance's strength is to bring quite independent classes together. In such cases, constructor conflicts should not occur.

Course Notes

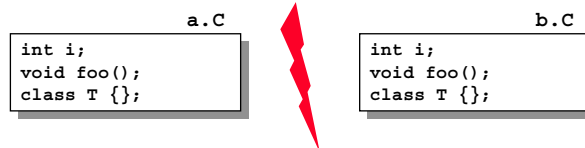


Chapter 18: Namespaces

- Namespaces
- Namespace Extension
- `using` Namespaces
- Namespace Nesting & Aliasing
- Name Lookup
- Koenig Lookup
- Interface Principle

Namespaces

- **The global namespace is easily polluted**
 - every global object, function, type or template lives there
 - every global entity must have a unique identifier
 - problematic when building code from disparate parts



Cannot be compiled and linked together, without first being modified

- **Namespaces introduce named global scopes**
 - reducing the opportunities for name collisions
 - helps structure the code into logical units

One of the goals of C++ is to promote code reuse through class libraries. Such libraries may capture common low-level mechanics (such as collection classes like STL), or business components. In the past few years, a large number of libraries have been made available by commercial software houses.

One problem that library developers face, along with developers of large C++ programs, is how to control the global name space. Specifically, how to avoid name space pollution. Every global entity must have a unique identifier in the global space, but it is not always easy to guarantee that this is the case. For example, how can a commercial software house be sure that the class names it produces will not clash with those of its customers? There may even be situations where different libraries have to use the same class names, for example, some implementations of CORBA have this requirement. The use of namespaces allows gateways between such libraries to be constructed.

Namespaces

```
#include <iostream>

namespace one {
    int i = 10;
}

namespace two {
    int i = 20;
}

int i = 30;

int main()
{
    int i;
    cout << i << " " << ::i << endl;
    cout << one::i << endl;
    cout << two::i << endl;
}
```

Namespaces qualify contained symbols with their name
– another example of name mangling

A namespace introduces a new named scope. To access entities within the namespace, the entity's name along with the namespace's name must be provided (using the scope resolution operator `::`). In other respects entities declared and defined within a user defined namespace behave as those declared within the global namespace. The global namespace is still, of course, accessible. Members of the global name space are accessed implicitly (in accordance with the usual C++ scoping rules) or explicitly by means of the unary scope resolution operator.

In the example all four variables are accessible from within `main` by use of the scope resolution operator along with the identification of the namespace.

Namespace Extension

- **namespace definitions are cumulative**
 - allowing separate declaration and definition entries

<pre>#ifndef STUFF_H #define STUFF_H namespace stuff { class Stuff { int j; public: Stuff (int = 0); operator int () const; }; class MinStuff : public Stuff { public: MinStuff (); operator int () const; }; } #endif</pre>	<p>stuff.h</p>
	<pre>namespace stuff { Stuff::Stuff (int i) : j(i) {} Stuff::operator int() const { return j;} MinStuff::MinStuff () : Stuff (10) {} MinStuff::operator int () const { return *this; } }</pre> <p>stuff.C</p>

If two namespaces are introduced with the same name, then the second namespace extends the first. The stuff namespace is first defined in stuff.h where two classes are defined. The namespace is then extended in stuff.C with the member function definitions of Stuff and MinStuff.

using Namespaces

- Introducing names into the current code region

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

int main()
{
    stuff::MinStuff a;
    std::cout << a << std::endl;
}
```

```
#include <stuff.h>

int main()
{
    using stuff::MinStuff;
    MinStuff a;
    // Stuff b;
}
```

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

using namespace std;

int main()
{
    using namespace stuff;
    MinStuff a;
    cout << a << endl;
}
```

Beware this form, as it returns
these names to the global
namespace

The using declaration provides a short-hand notation to identify namespace members by making their names directly accessible from the current code region. There are two distinct forms of the using declaration which have different effects on the scope of namespace members.

Namespace Nesting & Aliasing

- namespaces can be nested to arbitrary depth
 - each level is another scope

```
#include <iostream>

namespace outer {
    namespace inner {
        namespace in {
            int j = 11;
        }
    }
}

int main() {
    std::cout << outer::inner::in::j << std::endl;

    namespace oii = outer::inner::in;
    std::cout << oii::j << std::endl;
}
```

- Aliases are synonyms for namespaces
 - visible in the scope of the alias

Namespaces can be nested to any depth to help partition code. For example, an organisation may introduce an outer namespace, to avoid conflicts with code acquired externally. It may then partition the namespace based on functional or divisional boundaries. Nested namespaces may be extended by using the scope resolution operator to identify an inner level---so the physical file organisation need not be relevant to namespace management.

Using members of deeply nested namespace is sometimes cumbersome. To help simplify the code, aliases may be introduced for any namespace. The alias is simply a synonym for the namespace visible in the scope of its definition. A namespace may have any number of aliases, of which each may be used interchangeably.

Name Lookup

- Name lookup is necessarily subtle

```
namespace one
{
    class A;
    void foo ();
    void bar (A);
}

void foo (int i)
{
    foo (i);
}

void bar (one::A a)
{
    bar (a);
}
```

What happens here?

What happens here?

In the first example, foo is called recursively. The foo defined within namespace one is not in scope.

In the second example, the code will not compile. There is an ambiguity between bar calling itself, or bar calling the bar that is defined within namespace one. The reason for the ambiguity is known as *koenig lookup*.

Koenig Lookup

- **Compiler considers matches for the function ...**
 - in given function scope
 - in the namespaces of all argument types

```
#include <iostream>

namespace one
{
    class A {};
    void bar (A) {
        std::cout << "I need a bar" << std::endl;
    }
}

int main () {
    one::A a;
    bar (a);
}
```

```
$ ./a.out
I need a bar
```

```
<< versus std::<<
```

In the above code the compiler matches the `bar` function defined in namespace `one`, in spite of the fact that `bar` was not explicitly scoped as belonging to namespace `one` when it was called.

This behaviour may seem odd, it may even seem to violate the very idea of the namespace. Nevertheless, virtually every program written that uses the `<iostream>` makes use of Koenig lookup. Without it, the compiler would not know that operator `<<` is defined in namespace `std`, it looks there precisely because one of its operands (the left one) is an object of the `ostream` class which is defined in namespace `std`.

Interface Principle

- **A classes interface is defined by ...**
 - member functions
 - member data
 - class member functions and data
- **And ...**
 - any function supplied with the class than mentions the class

```
myString operator+ (const myString&, const myString&)
```

- is clearly meant to be part of the “myString” abstraction
 - but is a free (and friend) function for + argument symmetry
- **Namespace is a logical unit**
 - a collection of classes and functions designed to work together

Term coined by Herb Sutter in Exceptional C++.

Course Notes

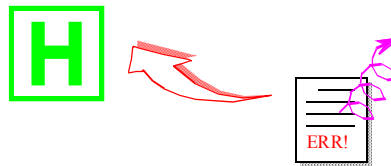


Chapter 19: Exception Handling

- Exception Handling Constructs
- Exceptional Nesting
- Re-Throwing Exceptions
- Exception Classes
- Catch All
- Resource Acquisition is Initialisation
- Function Try-Block
- Exception Specifications
- Writing Exception-Safe Code

Exception Handling

- **A mechanism for the controlled recovery of errors**
 - employs termination semantics, unwinding program to recover
- **Allows for separation of error handling responsibilities**
 - error detection (in a library) from recovery (in client code)



- **Much greater control than returning error codes**
 - constructors and functions return l-values can't return codes

Exception handling allows the detection of an error, which may occur deep within a reusable class library, to be separated from the handling of the error. Since error handling depends upon the circumstances of the client code, the client is best able to determine the relevant course of action to take.

Exception handling is particularly useful for notification of a failed constructor. Since constructors do not return values, the only way a client can determine that an error has occurred is to interrogate the defunct object. With exceptions, the constructor could have terminated the construction process, unwound any work which had been done (such as partial construction of members) and transferred control to an exception handler in the client code.

Another area where exceptions are useful is for functions which return modifiable l-values. That is, values which may appear on the left hand of an assignment, and be assigned to. For example, consider a user defined array class which supports an overloaded subscript operator [].

When the array subscript operator is given an index out of bounds what should it do? Returning an error code is pointless since the value returned is obviously a reference to the array's internal element storage--and the client, as in this case, may plan on assigning a new value into the referenced location. Without exceptions the operator could return a reference to a dummy variable, it could resize all indexes within range, or it could exit the program. None of these is particularly desirable, so good library design would probably allow the user to determine how the error is handled.

Exception Handling Constructs

- **Three language constructs to manage exceptions**

```
try { . . . }  
catch ( <object> )  
throw ( <object> )
```

try block delimits region of code aware of a set of exceptions

catch handlers follow each try block to intercept exceptions based on type

exception objects are thrown and code is unwound until suitable handler found

- **Uncaught exceptions handled by `unexpected()`**
 - typically causing the program to abort

The three language constructs supporting exception handling generally fall into two camps. `try` and `catch` are used on the client side to define exception enabled blocks of code and catch exception objects; `throw` usually resides on the library side ready to throw an exception object whenever an error occurs.

When `throw` raises an exception an object is thrown as the exception. The function in which the exception occurs, and all intermediate functions are collapsed (and the stack is unwound) until a catch handler is found able to cope with the exception object. That is, until the first handler which takes an object of a compatible type is found. Since an object is thrown (rather than simply an error code or message) any kind of information can be passed back to the catch handler. In fact, class libraries often have an associated exception class library, in which various exception classes are derived from each other. Since C++ unwinds the intermediate functions between `throw` and `catch`, it does not enable the client to directly resume execution in the library when the exception is resolved. However, care is taken during the unwinding process to ensure that objects created on the intermediate function stack frames are properly destroyed. It should therefore be possible for the client to restart the program after the handler.

Historical Note

`throw` was used in place of `raise` since the latter is used in C libraries. `signal` was considered but rejected for the same reason. `catch` was chosen since it is the logical counterpart to `throw`. However, there was some discussion of also using `catch` in place of `throw`, giving symmetry to the two sides of the exception mechanism. This was rejected as it was felt to introduce too much potential for error.

Exceptional Example

```
#include <iostream>

void f ()
{
    static int i = 0;
    if (++i < 10)
        throw i;
}

int main()
{
    for (;;)
    {
        try
        {
            f();
            break;
        }
        catch (int i)
        {
            std::cout << "exception " << i << std::endl;
        }
    }
}
```

control passed from throw to nearest
suitable handler, unwinding stack

execution continues after catch handler

```
$ ./a.out
exception 1
exception 2
exception 3
exception 4
exception 5
exception 6
exception 7
exception 8
exception 9
```

All code called directly or indirectly within the try block is sensitive to exceptions. When an exception is thrown (in this case an int object) all intermediate functions are unwound until an appropriate handler can be found. Having handled the exception the program continues after the catch block (or set of blocks).

Exceptional Nesting

stacks unwind until const char *
handler found in main

```
int main()
{
    for (;;)
    {
        try
        {
            f();
            break;
        }
        catch (int i)
        {
            std::cout << "exception " << i << std::endl;
        }
        catch (const char *s)
        {
            std::cout << "exception " << s << std::endl;
        }
    }
}
```

```
#include <iostream>

void g(int j)
{
    if (j%2 == 0)
        throw ("an even number");
}

void f()
{
    static int i = 0;
    while (++i < 10)
        g(i);
}
```

This example is similar to the earlier example except that it demonstrates that the try block holds for all code executed within the block, including nested functions. Also note that multiple catch handlers can be placed beneath the try block. In fact, following a try block is the only place where catch handlers can be defined. Irrespective of which handler is used, execution resumes after all of the handlers.

Re-Throwing Exceptions

```
int main() {
    for (;;) {
        try {
            f();
            break;
        }
        catch (int i) {
            std::cout << "exception " << i << std::endl;
        }
        catch (const char *s) {
            std::cout << "exception " << s << std::endl;
            throw;
        }
    }
}
```

```
#include <iostream>

void g() {
    throw ("weeee");
}

void f() {
    try {
        g();
    }
    catch (const char *) {
        throw;
    }
}
```

```
$ ./a.out
exception weeee
Abort (coredump)
```

In this example two try blocks are used, one nested within a function which is called within the first block. When the exception is thrown from `g()` it caught by the first suitable handler in `f()`. `f()` immediately re-throws the exception, and it is later caught in `main()`. The exception is again re-thrown but this time since no handler can be found the program aborts. This is the normal result in the event of there being no suitable handler.

When re-throwing the exception, the original exception object is thrown. If a derived object is caught by a base class catch handler and then re-thrown, it is the derived object that is re-thrown. If a handler wishes to change the object being caught (perhaps prior to re-throwing it), then the handler must take a reference to the exception object. In any case, references would often be used since they allow normal polymorphic behaviour to occur. An exception that is re-thrown may be caught by an enclosing try block, but not by another handler associated with the current try block.

Note that it is valid C++ to omit the name of a formal argument in a catch handler (just as it is with function declarations and definitions). In the catch handler in `f()`, only the type of the argument is specified. This is known as an anonymous parameter and is especially useful when the purpose of an argument is to select a particular overloaded function, rather than provide a value.

Exception Classes

```
class Matherr {};  
class Overflow : public Matherr {};  
class Underflow : public Matherr {};  
class DivideZero : public Matherr {};  
  
class BigInt {  
    long val;  
public:  
    BigInt (long l = 0) : val (l) {}  
    friend BigInt operator/ (const BigInt&, const BigInt&);  
    // ... other math functions  
};  
  
BigInt operator/ (const BigInt& l, const BigInt& r) {  
    if (r.val == 0)  
        throw (DivideZero());  
    return l.val / r.val;  
}
```

```
int main() {  
    try {  
        BigInt a (100), b (0);  
        a / b;  
    }  
    catch (DivideZero) {  
        std::cerr << "Divide Zero";  
    }  
    catch (Matherr) {  
        std::cerr << "Math Error";  
    }  
}
```

```
class NetworkFileError : public NetworkError, public FileError
```

In this example the exception class hierarchy serves no behavioural purpose since objects of the class have no data or functions. The purpose of the objects is simply to identify the type of error being thrown. By using inheritance, collections of related objects can be caught with a single catch handler defined for the base class (or indeed any parent class). In the example, if `BigInt` throws an `Overflow` this can be caught by the `Matherr` handler.

Usually, exception classes do contain functions and data so that they can pass information about the exception to the client code.

It is also possible to build exception hierarchies based on multiple inheritance. For example, consider the situation in which a program is interested in either network errors or file system errors.

```
class NetworkFileError : public NetworkError, public  
FileError
```

The useful thing about inheritance is that a `NetworkFileError` can be caught by either a handler designed to deal with `NetworkErrors` or one design to deal with `FileErrors`. This is the inheritance principle of contravariance and substitutability.

Catch All

- To catch any kind of exception use the ellipsis (...)

```
void allocate () {
    SomeResource r;
    r.acquire();

    try {
        // use r in various ways which may
        // give rise to an exception
    }

    catch (...) {
        r.release();
        throw;
    }
    r.release();
}
```

- **Partial exception handling**
 - releasing `SomeResource` then re-throwing

The ellipsis prototype is used to indicate any type, and may therefore be used to catch any exception object. A typical application is to support partial exception handling, where a local handler performs some immediate remedial action and then re-throws the exception for a more specific handler to catch. In this example, irrespective of the exception thrown, it is important to release the resource before the local stack unwinds.

Note that the try block could not have been placed around the declaration of the `SomeResource` object because this would have placed the variable `r` in a different scope to that of the exception handlers.

Resource Acquisition is Initialisation

- **RAII: resources are tied to the life-time of objects**
 - deterministic lifetime allows object to own resources

```
void allocate () {  
    SomeResource r;  
    r.acquire();  
  
    try {  
        //  
    }  
  
    catch (...) {  
        r.release();  
        throw;  
    }  
    r.release();  
}
```

Constructor acquires resource

Destructor releases resources,
called when object dies due to
normal return or exception.

Bjarne Stroustrup invented the design pattern of RAII. It guarantees that a resource cannot be used until an object is created and makes them available, as the constructor is responsible for resource acquisition. And it guarantees that resources are released when an object dies, whether or not from an exception causes the stack to collapse or an object normally ending its life. This is a significant advantage of C++ over languages with weaker guarantees of object lifetime (typically due to garbage collection schemes).

Function Try Block

- Allows entire body of a function to be exception aware

```
int main()
try {
    // code that may
    // throw an exception
}
catch (Exception1) {
    //
}
catch (Exception2) {
    //
}
```

Variables in function are not in scope of handlers

Only way of catching exceptions from constructor initialisation list

```
Employee::Employee (const String& n, Salary m, Dept& d)
try
    : wages (m), name (n)
{
    d.welcome(*this);
}
catch (...) {
    // handle any exception from constructor
}
```

A function try block allows an entire function to be placed within a try block. As shown in the example, in place of the braces that normally delimit the body of a function, a try block is used. The advantage of this is that the whole function is protected, and there is physically a cleaner separation between normal and exception code. In particular, the exception handlers are now placed outside and immediately after the function.

Since a try block introduces another scope, it is important to realise that none of the variables introduced in the function will be in scope in the catch handlers. This is one reason why try blocks are often placed within a function.

In the event that no handler is found, then the search for a suitable handler will continue in the calling function. In this example of a try block around main, if no handler is found then there is no appropriate handler in the program. Consequently `terminate()` is called, which (by default) will call `abort()` and the program will exit abnormally.

Constructor functions may have two parts, an initialisation list and a function body. Since exceptions may be thrown from either of these parts, and since normal try blocks are limited to the body of a function, function try blocks offer the only mechanism to catch all exceptions that may be thrown so that they can be handled by the constructor itself.

Exception Specifications

- **Used to list the exceptions that a function may throw**
 - guarantees that no other exceptions will be thrown

```
void foo () throw (Overflow);
void bar () throw (Overflow, DivideZero);
void any ();
void none () throw ();

void (*p)() throw (Overflow, DivideZero);
p = bar;
p = foo;

void (*j)() throw (Overflow);
// j = bar;
j = foo;
```

- **Exception violations at run-time terminate the program**
 - `unexpected()` is called which ultimately aborts the code

To enable the safe use of libraries it is important that all functions advertise the list of exceptions that they intend to throw. This is achieved by an exception specification which forms part of the prototype of the function. The specification lists the exact types of each object that may be thrown.

To indicate that a function does not throw any exceptions an empty argument specification list should be provided. In the absence of a throw specification, a function may throw any exceptions.

Since the throw specification is part of the prototype of a function, it must be used when defining pointers to functions. A pointer may, however, be less (but not more) restrictive than the function being assigned to it. In the example, functions addressed by `p` may throw `Overflow` or `DivideZero` exception objects. Since `foo` only throws `Overflow`, it can safely be assigned to `p`. A function which may throw some other exception (say, `RoundingError`) may not be assigned to `p`, since users of `p` would not know to anticipate this kind of exception object.

Writing Exception-Safe Code

- **Non-trivial, especially when mixed with templates**
 - cannot be retrofitted, design must consider exception safety
- **Some basic rules ...**
 - never let an exception escape from a destructor or `delete`
 - structure code to remain consistent if an exception occurs
 - correctly release resources if an exception occurs
 - propagate unhandled exceptions to caller
- **Guaranteeing exception safety**
 - **Basic:** don't leak resources
 - **Strong:** employ commit-or-rollback semantics on state
 - **Nothrow:** function will never emit an exception

Writing exception safe code is considerably more complex than it first appears. Entities in C++ (functions, classes, etc) are often dealing with other entities whose types (as they are templates) and behaviour (as its hidden elsewhere) is unknown. Consequently, it is possible that exceptions may arise at the most inconvenient times, causes your code to leaking resources, and perhaps worse, to leave an object state inconsistent.

The guarantees listed above are from Dave Abrahams. For a detailed discussion see *Exceptional C++* by Herb Sutter.

Chapter 20: Templates

- **Template Functions**
 - **Template Classes**
 - **A Template Stack**
 - **Using the Template Stack**
 - **Non-Template Type Parameters**
 - **Recursive Composition**
 - **Inheriting from Templates**
 - **Default Template Parameters**
 - **Template Specialisation**
 - **A Modern C++ Program**
-

Templates

- **Define families of functions and classes**
 - parameterised on argument and member types



define the behaviour and state of the idea of a bottle independently of the type of what it contains

- **Provides the foundation for generic programming**
 - a foundation of the C++ Standard Library and modern C++
- **Templates are evaluated at compile time**
 - often results in highly efficient code, though sometimes bloated

Templates provide a compile time mechanism in which types used by functions and classes are parameterised. Programmers are required to give only one generic version of such functions and classes. The compiler will then generate whatever concrete versions are required in the program. For example, having defined a generic swap function which operates on two parameters, the compiler will generate as many versions of this function as are necessary to handle the different usages in the program.

Essentially, templates are a meta-programming language; a language in which blue prints for functions and classes may be defined, rather than concrete functions and classes which could actually be used. The blue prints are used by the compiler as it generates code on demand, based on the requirements of a C++ program.

The costs of templates are longer compilation times and larger programs. The former is a result of the extra work which the compiler must do, but at least the programmer is saved the burden of recording the same thing. The latter is because it is easy for programmers to write small (apparently simple) programs which when expanded by the compiler result in massive programs. Also, some template algorithms have function calls 'unrolled' at compile time resulting in fast execution times, but large programs.

Template Functions

- **min is a template function**
 - template is instantiated to make a function for each type of use

all part of one
definition

```
#include <iostream>

template <class T>
T min (T a, T b) { return a < b ? a : b; }

int main()
{
    int a = 10, b = 20;
    char d = 'A', e = 'Z';

    std::cout << min (a,b) << std::endl;
    std::cout << min (d,e) << std::endl;
    // std::cout << min (a,e) << std::endl;
}
```

- **What assumptions does min place on objects of T?**

A template function is a function in which at least one piece of type information (a local variable or a formal argument) is parameterised. In the example, function min is parameterised on template type T. The declaration introduces template variable T for the entire scope of the following function. The concrete type that T evaluates to is not important until the function is actually used. As far as the definition of min is concerned, T may be used just like any other type.

In the example, min takes two formal arguments and returns a single value. Although the concrete type of T of these arguments and values is unknown, it is clear that they are all of the same type. The body of min is straightforward, although it does suggest an assumption made about T. That is, that operator < is defined for the type. Since T is in scope for the entire function, it is also possible to introduce local variables of type T inside the function.

Two calls to min are made in the main program; the first passes two int variables and the second two chars. When encountered by the compiler (or pre-linker), the specific versions of min required to satisfy the function calls are built. This is known as template instantiation. An important detail to note here, is that the compiler was able to deduce the type of T based upon the types of arguments supplied during each function call.

If min is called with different types of arguments, as in the commented out case, then the compiler is unable to instantiate min. Should the T be made into a char or an int?

Template Classes

- **Each template class defines a family of classes**
 - used extensively in collection classes

```
#ifndef DATA_H
#define DATA_H

template <class T>
class Data
{
    T i;
public:
    void set (T);
    T get () const;
};

template <class T>
void Data<T>::set (T value) { i = value; }

template <class T>
T Data<T>::get () const { return i; }

#endif
```

```
int main()
{
    Data<int> d;
    d.set (10);
    int j = d.get ();
}
```

Class data is parameterised on template type T. It is a blue print rather than a concrete instantiable class. For each distinct value of T, a new instance of the class can be instantiated and used to produce objects.

Unlike template functions, the template parameter T must be explicitly specified by the users of the class. In the example code, an int version of class Data is requested. This causes the compiler (or pre-linker) to make an int version of Data, so that object d can be instantiated from the class.

The name of the class instantiated by the compiler, in this example, is Data<int>. The names of the member functions are Data<int>::set and Data<int>::get. The reason why the syntax of the code is complex, is that each member function is itself a template function. Ultimately, there will be many versions of get and set, belonging to the different instantiations of Data. The reason why template<class T> must be restated so often, is that it merely introduces the template variable T for the scope of the following function or class.

A Template Stack

```
template <class T> class Stack
{
    T *elements;
    int size;
    int current;
public:
    Stack (int);
    ~Stack ();
    void push (const T&);
    T pop ();
    int isEmpty();
    int isFull();
};
```

there is nothing to compile until template is instantiated

compiler needs a **T** to generate code

code typically placed in header file with compilation guard

```
template <class T>
Stack<T>::Stack (int s)
    : size (s), current(0), elements (new T [s]) {}

template <class T>
Stack<T>::~~Stack () { delete [] elements; }

template <class T>
void Stack<T>::push (const T& e) { elements
[current++] = e; }

template <class T>
T Stack<T>::pop () { return elements [--current]; }

template <class T>
int Stack<T>::isEmpty() { return current == 0; }

template <class T>
int Stack<T>::isFull() { return current == size; }
```

This example illustrates the power of generic programming with templates. Once the template is defined, describing the logical data structure of a stack, it need not be defined again. Whether the user requires a stack of ints, a stack of Accounts or a stack of bananas, the underlying data structure is exactly the same. Following the template definition, the compiler is able to generate as many kinds of stack as are required. Essentially, templates allow the separation of program logic from those things it is being applied to.

The Stack class is a full example of a template class. A stack is a fairly simple data structure providing first-in, last-out access semantics. The Stack class shown is quite straightforward and only differs from a conventional class by the template keyword and use of the parameterised type `<T>`. The constructor for the Stack will dynamically allocate an array of Ts and store the address of the first in elements. size records the total number of elements able to be stored. current is a pointer moving up and down the stack to facilitate pushing and popping.

Using the Template Stack

- Given a template type, compiler builds a class
 - compiler cannot deduce desired type as with template functions

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

main()
{
    Stack<int> q(10);
    q.push (3);
    q.push (6);
    q.push (8);
    std::cout << q.pop () << std::endl;
    std::cout << q.pop () << std::endl;
    std::cout << q.pop () << std::endl;

    Stack<char *> r(3);
    r.push ("hello");
    r.push ("world");
    std::cout << r.pop () << std::endl;
    std::cout << r.pop () << std::endl;
}
```

```
$ ./a.out
8
6
3
world
hello
```

In this program the compiler is required to generate two versions of `Stack<T>`. One to hold integers (`Stack<int>`), and one to hold character pointers (`Stack<char *>`). Having constructed the classes, code is generated to allow the run-time instantiation of the objects `q` and `r`.

Non-Template Type Parameters

- **Allows values to be passed at compile time**
 - useful with template meta-programming

```

#ifndef STACK_H
#define STACK_H

template <class T, int i>
class Stack
{
    T elements[i];
    int size;
    int current;
public:
    Stack ();
    ~Stack ();
    void push (const T&);
    T pop ();
    int isEmpty();
    int isFull();
};

#endif

```

```

template <class T, int i>
Stack<T,i>::Stack () : size (i), current(0) {}

template <class T, int i>
Stack<T,i>::~~Stack () {}

```

Stack now implemented using a fixed array

A new class is generated for every new type and size!

Not all template arguments have to be types. It is possible and often useful to pass concrete values as template arguments which are subsequently used in the construction of the class. Since template instantiation occurs at compile time the values must of course be available at compile time.

In this example the Stack class is parameterised on the template type T and an int i. In order for the template class to be instantiated, the compiler must be supplied a concrete type for T and a value for i. In the Stack, T is used as the element type whilst i is used to determine the length of the array elements. The new class is similar to previous versions, except that the element array is no longer allocated dynamically. As shown, both the constructor and destructor functions are much simpler.

The reason why i is passed in as a template argument is that C++ arrays are sized at compile time. Their size can therefore only be based on a constant expression. Since the programmer may require different size stacks, the only way of parameterising this constant expression, is via a template.

In the following code fragment an instance of the template class is generated for type int with an element array length of 10. q is an object of the class and so can store up to 10 integers.

```

int main() {
    Stack<int,10> q;
    q.push (3);
    q.push (6);
    std::cout << q.pop () << std::endl;
    std::cout << q.pop () << std::endl;
}

```

Recursive Composition

- Nesting to create collections of collections of ...

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

main()
{
    Stack<char *,3> r,j;
    r.push ("hello");
    r.push ("world");
    std::cout << r.pop () << std::endl;
    std::cout << r.pop () << std::endl;

    Stack <Stack <char *, 3>,5> ok;
    ok.push (r);
    ok.push (j);
}
```

maximal munch

```
Stack <3, Stack <5, char *>>
Stack <3, Stack <5, char *> >
typedef Stack <5, char *> stack0char
Stack <3, stack0char>
```

When instantiating templates the instantiation type may itself be an object from a template class. This gives rise to template nesting and is an extremely powerful programming technique. In this example, a stack of stacks is produced.

Template nesting sometimes gives rise to excessive compilation times, and may contributed to code bloat. However, as compilers get smarter this problems is being resolved. For example, the *egcs* compiler needed 200MB to compile a complex nested template.

Take care with C++ syntax. The 'maximal munch' problem is caused by the >> characters being interpreted as the right shift operator rather than a nested template parameter specifier. The solution is to add spaces. These types of problems are often avoided by using typedef to define names for complex types.

Inheriting from Templates

- **Templates may be used as parent or child class**
 - useful in policy based programming

```
#include <vector.h>

template <class T>
class SafeVector : public Vector<T>
{
public:
    SafeVector (int);
    const T& operator [] (int) const;
};

template <class T>
SafeVector<T>::SafeVector (int i) : Vector<T> (i) {}

template <class T>
const T& SafeVector<T>::operator [] (int i) const
{
    if (i < 0 || i >= getSize())
        throw ("Subscript out of range");
    return Vector<T>::operator [] (i);
}
```

Deriving a new template class from an existing one is relatively straightforward, requiring an indication of what type of base class the new class should be specialised from. This type indication may be a further parameterisation or concrete. In the above example `SafeVector` is derived from `Vector` (both being of type `T`). In `SafeVector` the subscript operators are overloaded to perform bounds checking. In the event of an indexing error (as below) the subscript operators from `SafeVector` will throw an exception.

An important limitation of this particular piece of code is that it inhibits the substitution of instances of the derived class in situations where instances of the base are expected. The throw in the new derived class would not be anticipated by base class code.

Default Template Parameters

- **Avoids the need to explicitly provide all types**

```
#include <iostream>

template <class A, class T = int> class Number {
    T num;
public:
    Number (T n) : num(n) {}
    void display() const {
        std::cout << num << std::endl;
    }
};

int main() {
    Number <double, double> num1 (3.14);
    num1.display();
    Number <int> num2 (3);
    num2.display();
}
```

- **Widely used in the C++ Standard Library**
 - default policies such as how to compare or allocate objects

Default template parameters provide missing type arguments should the client code not wish to explicitly specify them. They are commonly used to provide default policy (and therefore behaviour) for classes implementing the strategy design pattern through templates.

The technique of supplying a policy through a template argument and then defaulting that argument to supply the most common policy is widely used in the standard library.

In a similar fashion to function name overloading, template function overloading may also be used to allow specific methods to be selected based on the number of arguments provided.

Template Specialisation

- **Override general behaviour of template family**
 - useful when general solution inappropriate for specific type
 - partial template specialisation powerful

```
#include <iostream>
#include <cstring>

template <class T> T max (T a, T b) {
    return (a > b ? a : b);
}

typedef const char * CP;
template<> CP max<CP> (CP a, CP b) {
    return (strcmp (a, b) > 0 ? a : b);
}

int main() {
    int i=10, j=5;
    std::cout << max (i,j) << std::endl;
    const char *x = "one";
    const char *y = "two";
    std::cout << max (x,y) << std::endl;
}
```

Template specialisation allows the general template definition to be overridden for specific types. It is particularly useful when the general behaviour in the template class or function is inappropriate for the type.

In this example, the max template function returns the maximum of its two arguments. Without the specialisation, when supplied with the two pointer arguments, it would return the maximum of the pointers rather than the maximum of the char * strings. The definition of the explicit template specialisation function may be simplified in the following way if the template argument can be deduced from the function parameters.

```
template<> CP max (CP, CP)
```

A Modern C++ Program

- Extensive use of templates, generics, policies, traits
 - as exemplified by the C++ Standard Library

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
#include <string>

using namespace std;

int main()
{
    istream_iterator<string> is (cin);
    istream_iterator<string> eof;

    vector<string> data;
    copy (is, eof, back_inserter (data));
    sort (data.begin(), data.end());

    ostream_iterator<string> os (cout);
    copy (data.begin(), data.end(), os);
}
```

Looks like a new
language!

The program reads a series of white-space separated words from the standard input until end-of-file, and then writes them out sorted.

A

Appendix A: Debugging with gdb

- **Getting Started with gdb**
- **Abnormal Terminations**
- **Viewing Source Code**
- **Breakpoints**
- **Monitoring Variables**
- **Watchpoints**
- **Postmortem Analysis with gdb**
- **Attaching to a Running Process**
- **Working with Threads**
- **Visual Debugging with DDD**

The GNU Debugger

- **`gdb`**
- **One of the earliest GNU tools**
 - development followed gcc compiler
- **Command line based debugging**
 - controlled execution of programs
 - post-mortem analysis
 - attaching to active processes
- **Often used with GUI frontends**
 - `xxgdb`
 - `ddd`

The GNU debugger, `gdb`, is perhaps the most widely used debugging tool in Linux systems today. It has evolved over a number of years into a highly sophisticated debugger.

`Gdb` supports the controlled execution of programs including single stepping, breakpoints and examination of data. It also allows core files generated by program crashes to be analysed to determine the cause of the problem. `Gdb` can also attach itself to an already active process, allowing further execution to be controlled. It can work with single or multi process applications, and even has support for multi-threaded programs.

Although the debugging operations offered by `gdb` are very powerful, one thing that has remained the same over the years is its user interface. `Gdb` is a command line oriented tool, in the true traditions of Unix. This can discourage developers who have been used to GUI based debuggers from using it. However there are alternative approaches – tools that use `gdb` as an underlying debugging "engine", and yet provide graphical interfaces.

An early attempt at this was the tool `xxgdb`, where a very simple X based user interface was added to `gdb`. This proved not to be a huge success, indeed most developers preferred the command line option.

However, more recently a graphical debugger called `ddd` has evolved from work done as part of a Master Thesis in a German university. `Ddd` is altogether a better product, with a much more powerful and easy to use user interface. We will discuss `ddd` in more detail later in this chapter.

Getting Started with gdb

- **Programs should be compiled with debugging support**
 - use `-g` flag
- **Optimisation should be disabled**
 - can cause problems
 - use `-O0` flag to disable all optimisations

```
$ gcc -g -O0 -o hello hello.c
$ gdb hello
GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
...various messages...
This GDB was configured as "i586-suse-linux"...
(gdb) run
Starting program: /home/george/dblinux/cprog/hello
Program exited normally.
```

Assuming an exit status of 0

As with most debuggers, gdb requires additional information to be embedded into a program for it to work effectively. The gcc compiler adds this information when it is called with the `-g` option. This is consistent with most Unix compilers.

Compiler optimisations can often cause problems for debuggers, since they can make it hard to relate compiled code back to source. When debugging a program it is always advisable not to use any compile time optimisations. With the gcc compiler, we actually need to be explicit in our request not to optimise, since the compiler has some optimisations that it will apply even if the developer does not ask for them (by using the `-O` flag). To disable all such processing, it is necessary to use the flag `-O0` (zero) in the compile operation. Although the difference may be insignificant in most cases, it is still worth remembering.

Once the program has been compiled, gdb can be invoked and told to read the required executable. From here, it is possible to start the program running using gdb's `run` command. Assuming there are no problems, the program executes, its standard output being shown within the output stream of gdb itself.

If the program exits with an exit status of 0, the message

"Program exited normally"

is displayed. If the exit status is non-zero then its value is displayed since this usually (but not always!) indicates a problem of some sort.

Abnormal Terminations

- Termination due to a signal
 - user provided (e.g. interrupt)
 - system provided (e.g. memory problem)

- Details of signal shown

```
$ gdb badptr
...
(gdb) run
Starting program: /home/george/progs/badptr

Program received signal SIGSEGV, Segmentation fault.
0x004010e4 in main (argc=1, argv=0xa042798) at badptr.c:8
8      printf("The answer is %d\n", *ip);
(gdb)
```

```
$ gdb loop
...
(gdb) run
Starting program: /home/george/progs/loop

Program received signal SIGINT, Interrupt.
(gdb)
```

If a program terminates as a result of being signalled, the termination is classed as *abnormal*. The signal may have occurred as the result of a user action, for example using Ctrl-C to break a program that may be caught in a loop. It may also occur due to an exceptional condition detected by the operating system, such as attempt to access a bad memory address.

Gdb will detect abnormal terminations and display details, as shown on the slide. In the event of an "exception" signal, if possible the source code line that caused the signal will be displayed.

Viewing Source Code

- **list command**

- shows "next" 10 lines
- can specify line numbers explicitly

```
(gdb) list
1      #include <stdio.h>
2      #include <fcntl.h>
3
4      char data [] = "Hello, world\n";
5
6      int main ( int argc, char *argv[] )
7      {
8          int fd = open ("Result", O_RDWR|O_CREAT, 0644);
9          if ( fd < 0 ) {
10             perror("Result");
(gdb) list 1,3
1      #include <stdio.h>
2      #include <fcntl.h>
3
(gdb)
```

Gdb displays lines from the appropriate source file using the *list* command.

With no arguments, this command will display the "next" 10 lines (the notion of a current line is maintained by gdb). A comma separated pair of arguments is treated as a starting and ending line for the display. A single argument indicates that 10 lines "around" the argument are to be displayed (i.e. 5 lines before and 5 lines after the specified line).

"Lines" may be specified by their number, optionally qualified by a filename, e.g.

`func.c:200`

It is also possible to provide a function name, which is interpreted as the first line of the function definition. This may also be qualified by a file name:

`func.c:bigFunc`

Execution

- **Use run command to run program from start**
 - can be abbreviated to r
- **Supply command line args after run command**
 - use set environment command to alter environment variables

```
(gdb) run a1 a2 a3
Starting program: /home/george/dblinux/cprog/args a1 a2 a3
arg 0: /home/george/dblinux/cprog/args
arg 1: a1
arg 2: a2
arg 3: a3
Program exited normally.
```

args.c

```
#include <stdio.h>
int main ( int argc, char *argv[] )
{
    int i;
    for ( i=0; i<argc; i++ ) {
        printf("arg %2d: %s\n", i, argv[i]);
    }
    exit(0);
}
```

Once loaded into gdb, a program is started using the command
run

Execution begins at the start and, unless otherwise instructed, proceeds to the end. gdb assumes an exit status of 0 means success.

If there are command line arguments to be passed to the program, these are supplied after the run command.

Breakpoints

- **Set using the break command**
 - use line number or function name
- **Clear using clear command**
 - `tbreak` command automatically deletes breakpoint after first stop

```
(gdb) break 13
Breakpoint 1 at 0x804848e: file hello.c, line 13.
(gdb) run
Starting program: /home/george/dblinux/cprog/hello

Breakpoint 1, main (argc=1, argv=0xbffff4e4) at hello.c:13
13      write(fd, data, strlen(data));
(gdb) info breakpoints
Num Type      Disp Enb Address      what
1  breakpoint keep y   0x0804848e in main at hello.c:13
    breakpoint already hit 1 time
(gdb) clear 13
Deleted breakpoint 1
(gdb)
```

Clear breakpoint at line 13

Breakpoints are set using the *break* command. The argument may be a line number or a function name, in which case the break occurs as the function is called.

To *clear* a breakpoint, use the same information that was used when it was set.

If the breakpoint was set using the *tbreak* command, it is automatically cleared after the first stop on the breakpoint. In other words it allows "once only" breakpoints.

While remaining set, a breakpoint may be enabled and disabled. Use the *enable* and *disable* commands, with the breakpoint number as its argument (not its line number as used in the set and clear commands).

Data

- **Use print command**

- supply variable name
- capable of evaluating expressions

```
(gdb) what's fd ←
type = int
(gdb) print fd
$1 = 7
(gdb) print data
$2 = "Hello, world\n"
(gdb) print strlen(data)
$3 = 13
```

From breakpoint at line 13
as in previous example

Change descriptor used for
output, so message goes to
stdout rather than to file

- **Use set variable command
to change value of variable**

- capable of expression
evaluation

```
(gdb) set fd=1 ←
(gdb) continue
Continuing.
Hello, world

Program exited normally.
(gdb)
```

gdb provides powerful features for manipulating data within a program. The slide shows some of these.

The examples assume the program from before, stopped at the breakpoint on line 13 that was set up on the previous slide.

We can examine any variable in scope at this point – find out its type and value. The print command is also capable of evaluating expressions, including the calling of functions that are available to the code at the point where execution is currently suspended.

An even more useful feature is the capability to alter the value of variables – in the example we change the value of the int variable "fd" from its value of 7 (which was set through the call to open()) to 1. Once we have done this, we can continue with execution and the write() call that immediately follows will send the data to this descriptor (i.e. to stdout) rather than to the previous one that related to the file.

Monitoring Variables

- **gdb maintains automatic display list**
 - expressions displayed each time program stops
 - shows how value of a variable is changing (or not)

```
(gdb) break 8
Breakpoint 2 at 0x8048454: file hello.c, line 8.
(gdb) run
Starting program: /home/george/dblinux/cprog/hello

Breakpoint 2, main (argc=1, argv=0xbffff4e4) at hello.c:8
8      int fd = open ("Result", O_RDWR|O_CREAT, 0644);
(gdb) display fd
1: fd = 1073823776 ← fd not set here
(gdb) continue
Continuing.

Breakpoint 1, main (argc=1, argv=0xbffff4e4) at hello.c:13
13     write(fd, data, strlen(data));
1: fd = 7 ← fd value is changed
(gdb)
```

gdb is capable of tracking the value of variables using its *automatic display list*. In fact, there is more flexibility than simply tracking variables. Each entry in the list can be an expression that is evaluated whenever the program stops, and its result displayed.

In the slide, we can see that at the first breakpoint the variable `fd` has not been assigned. The value has changed at the second breakpoint.

Continuing Execution after Breakpoints

- Use **continue** to proceed to next stop point
 - or end of program
- Single step execution using **step** command
 - use **next** if function calls to be skipped
 - use **stepi** and **nexti** for machine instruction level stepping

```
(gdb) display fd
1: fd = 6
(gdb) run
Starting program: /home/george/dblinux/cprog/hello

Breakpoint 1, main (argc=1, argv=0xbffff4d4) at hello.c:8
8         int fd = open ("Result", O_RDWR|O_CREAT, 0644);
1: fd = 1073823776
(gdb) step
9         if ( fd < 0 ) {
1: fd = 6
(gdb)
```

After stopping at a breakpoint, execution may be resumed from the current position using the *continue* command. The program will now run until the end (or until the next breakpoint).

gdb also allows the program to be run one instruction (one source line) at a time, using the *step* command. If the next statement is a call to a function, then this will stop at the first instruction of the function. To step "over" function calls use the *next* command.

Both *step* and *next* have equivalent commands that operate at the level of machine instruction rather than source line – use *stepi* and *nexti* for this.

Watchpoints

- Like breakpoints, but based on expressions
 - not locations in program
 - "break when the value of this variable changes"

```
(gdb) run
Starting program: /home/george/dblinux/cprog/hello

Breakpoint 3, main (argc=1, argv=0xbffff4d4) at hello.c:8
8      int fd = open ("Result", O_RDWR|O_CREAT, 0644);
(gdb) watch fd
Hardware watchpoint 5: fd
(gdb) continue
Continuing.
Hardware watchpoint 5: fd

Old value = 1073823776
New value = 6
main (argc=4, argv=0xbffff4d4) at hello.c:9
9      if ( fd < 0 ) {
(gdb) continue
```

Watchpoints are like breakpoints, but are based on the value of an expression changing rather than a specific location in the program. This can make monitoring variable values more straightforward.

A watchpoint is specified with an expression. We can specify that the break is to occur whenever the expressions value is read by the program, using *rwatch* or when it is changed, using *awatch* or the default which is to monitor for read or write *watch*.

Postmortem Analysis with gdb

- Supply core file name with program name

```
annet$ badprog
Segmentation fault (core dumped)
annet$ gdb badprog core
GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
...
This GDB was configured as "i586-suse-linux"...
Core was generated by `badprog'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x08048365 in main (argc=1, argv=0xbffff4f4) at badprog.c:6
6   i = *ip
(gdb)
```

gdb can be used to discover why a program has failed – for this we make use of the core file that was generated as a result of the program fault. Most runtime exception signals cause a core file to be written – it contains an image of the memory that the program was using when it failed.

It may be necessary to modify a resource limit setting in the shell to allow core files to be written - the shell is often configured to place a limit of 0 on the size of core file that can be created. To override this use the command

ulimit -c unlimited

(or you can specify a maximum size, in 512 byte disk blocks, or core file).

Note also that core file will not be written if the failing process does not have write permission in its current directory.

If there is a core file, we start **gdb** with its name as an additional argument. As part of the initialisation sequence, **gdb** will print a brief indication of where the failure happened.

What Happened?

- Use `where` command to get stack trace
 - or `backtrace` command

```
(gdb) where
#0  0x08048365 in main (argc=1, argv=0xbffff4f4) at badprog.c:6
#1  0x4003e8ae in __libc_start_main () from /lib/libc.so.6
(gdb) info locals
i = 0
ip = (int *) 0x0
(gdb) info args
argc = 1
argv = (char **) 0xbffff4f4
(gdb) list 6
2
3      int main ( int argc, char *argv [] )
4      {
5          int i=0, *ip=0;
6          i = *ip;
7          printf("The value of i is %2d\n", i);
8      }
```

```
#include <stdio.h>
int main ( int argc, char *argv [] )
{
    int i=0, *ip=0;
    i = *ip;
    printf("The value of i is %2d\n", i);
}
```

Key to discovering what happened, and how it happened, is the ability to obtain a call history of the program. We will have been told *where* the program failed, but not *how* it came to be executing that particular piece of code.

The *where* command, or *backtrace* command will display the stack for the process. Local variables for the current stack frame are displayed using *info locals* and function arguments, if appropriate with *info args*.

In this example it is clear what has caused the problem – an attempt to dereference the pointer *ip* when it has value 0.

Note that sometimes stack frames will not have debugging information available, for example if they are standard library functions that were not compiled with the `-g` flag. This can make analysing core files somewhat more difficult, though not impossible...

Attaching to a Running Process

- Use `attach` command

```
annet$ loop &
[1] 19127
annet$ gdb
GNU gdb 5.3
...
(gdb) attach 19127
Attaching to process 19127
Reading symbols from /home/george/dblinux/cprog/loop...done.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0x08048339 in main (argc=1, argv=0xbffff504) at loop.c:11
11         i+=4;
(gdb) s
9         int i = 3;
(gdb) where
#0  main (argc=1, argv=0xbffff504) at loop.c:9
#1  0x4003e8ae in __libc_start_main () from /lib/libc.so.6
(gdb)
```

`gdb` is able to attach itself to an already running process. Use the *attach* command, supplying the process id of the target process. Of course, for the output to be meaningful the process will have to be executing a program compiled with the debug flag `-g`.

Once attached, `gdb` stops the program. From there, all `gdb` commands are available to examine data, control execution, insert break points and so on.

To terminate the child process, use the *kill* command. To detach the debugger from the process, leaving it to continue "unhindered", use the *detach* command.

Multithreaded Programs

- **gdb is capable of debugging multithreaded programs**
 - assuming underlying system has necessary support

```
$ gdb phil
...
(gdb) break think
Breakpoint 1 at 0x804852a: file phil.c, line 10.
(gdb) run
Starting program: /home/george/dblinux/cprog/phil
[New Thread 16384 (LWP 19588)]
[New Thread 32769 (LWP 19589)]
[New Thread 16386 (LWP 19590)]
[New Thread 32771 (LWP 19591)]
[Switching to Thread 32771 (LWP 19591)]
Breakpoint 1, think () at phil.c:10
10      sleep(2);
(gdb) c
Continuing.
[Switching to Thread 16386 (LWP 19590)]
Breakpoint 1, think () at phil.c:10
10      sleep(2);
(gdb)
```

On Linux systems that support threads, gdb is perfectly usable with multithreaded programs.

The slide shows an example of such a program – an implementation of the Dining Philosophers problem, where a breakpoint is introduced in the function think.

As each new thread is created, gdb reports this fact. When execution switches between threads, it is also reported.

Working with Threads

```
(gdb) info threads
[New Thread 49156 (LWP 19592)]
  5 Thread 49156 (LWP 19592)  0x400a2768 in sigsuspend () ...
  4 Thread 32771 (LWP 19591)  0x4012a166 in nanosleep () ...
* 3 Thread 16386 (LWP 19590)  think () at phil.c:10
  2 Thread 32769 (LWP 19589)  0x40034470 in ...
  1 Thread 16384 (LWP 19588)  0x400a2768 in sigsuspend () ...
(gdb) where
#0  think () at phil.c:10
#1  0x08048561 in philosopher (number=0xbffff48c) at phil.c:24
#2  0x4002ec60 in pthread_start_thread () from ...
#3  0x4002ecdf in pthread_start_thread_event () ...
(gdb) thread 5
[Switching to thread 5 (Thread 49156 (LWP 19592))]#0  0x400a2768
                                in sigsuspend() from /lib/libc.so.6
(gdb) where
#0  0x400a2768 in sigsuspend () from /lib/libc.so.6
#1  0x40031198 in __pthread_wait_for_restart_signal ()
    from /lib/libpthread.so.0
#2  0x4003290c in __pthread_lock () from /lib/libpthread.so.0
#3  0x4002eccc in pthread_start_thread_event () ...
```

Information about currently active threads can be obtained using the *info threads* command. The thread(s) that is(are) currently executing are highlighted using the '*' character. Any request on stack information, local variables or arguments will relate to this thread.

To change the "current thread" use the *thread* command, giving the thread number as listed in the output from the *info threads* command. Notice that now the *where* command relates to this thread.

Graphical Debugging with DDD

- **Graphical Debugger**
 - part of GNU project
 - <http://www.gnu.org/software/ddd>
- **Operates with gdb**
 - provides GUI "wrapper"
- **All gdb commands available**
 - data structures can be graphically displayed
- **Also works with other command line debuggers**
 - dbx
 - jdb



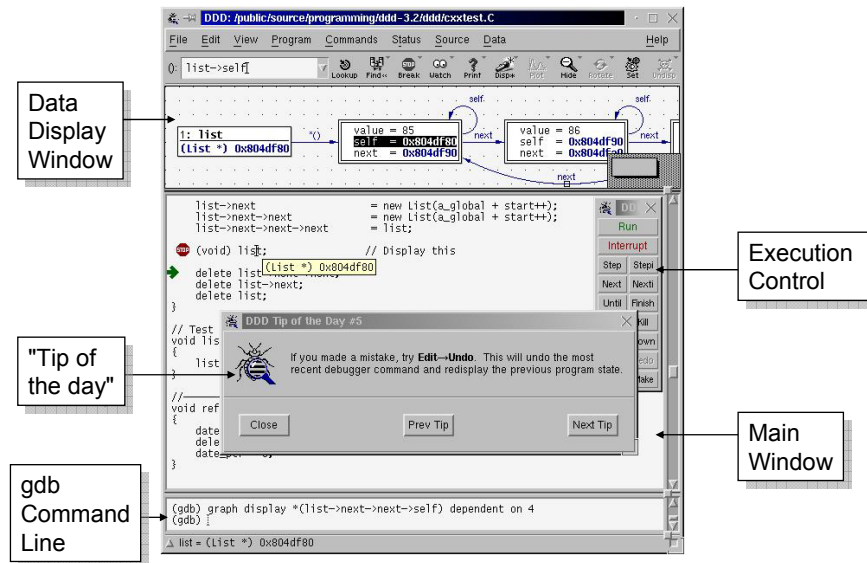
While gdb is an extremely powerful debugger, its command line interface is nevertheless somewhat old-fashioned. Over the years there have been numerous attempts to provide GUI front-ends for gdb, such as xgdb.

However one of the most successful tools in this area is ddd.

ddd began as a student project in a German university, but has proved to be so useful that it is now officially kept under the GNU project.

It operates by providing a GUI front-end that communicates with the actual debugger (known as the *inferior* debugger) through a pseudo-tty interface. ddd can operate with other inferior debuggers, so long as they have a command line interface. For example it is able to use dbx and the JDK debugger jdb.

Using DDD

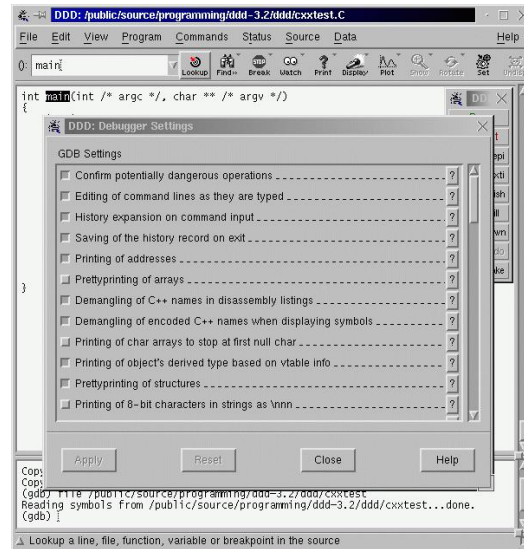


The slide shows the basic window arrangement of ddd. The main window contains a display of the source code that is currently being worked on. Breakpoints and current position are shown here as well. Below this is a command window, providing access to all the normal commands of the inferior debugger. This is useful because there are some infrequently used commands that are not made available through the button/mouse route.

Perhaps the strongest feature of ddd is its ability to display data structures graphically – this is done in the top window. If the data structures are being watched (using display) then this diagram will automatically update. Linked data structures are shown as such, and field values displayed.

Configuration of gdb Through ddd

- Many gdb properties configurable from ddd



gdb is a highly configurable debugger, and ddd allows access to most of the configuration properties through its settings options.

Summary

- **Linux applications request services from the kernel using system calls**
 - may be hidden behind library function calls
 - **The strace command allows system call activity to be monitored**
 - **Library calls may be traced using ltrace**
 - **gdb is the standard Linux debugger**
 - command line oriented
 - full debugging capabilities
 - supports multithreaded programs
 - **Graphical debugging is possible using dd**
 - wrapper around gdb (or other debugger)
-



Course Notes



Course Notes



Course Notes

