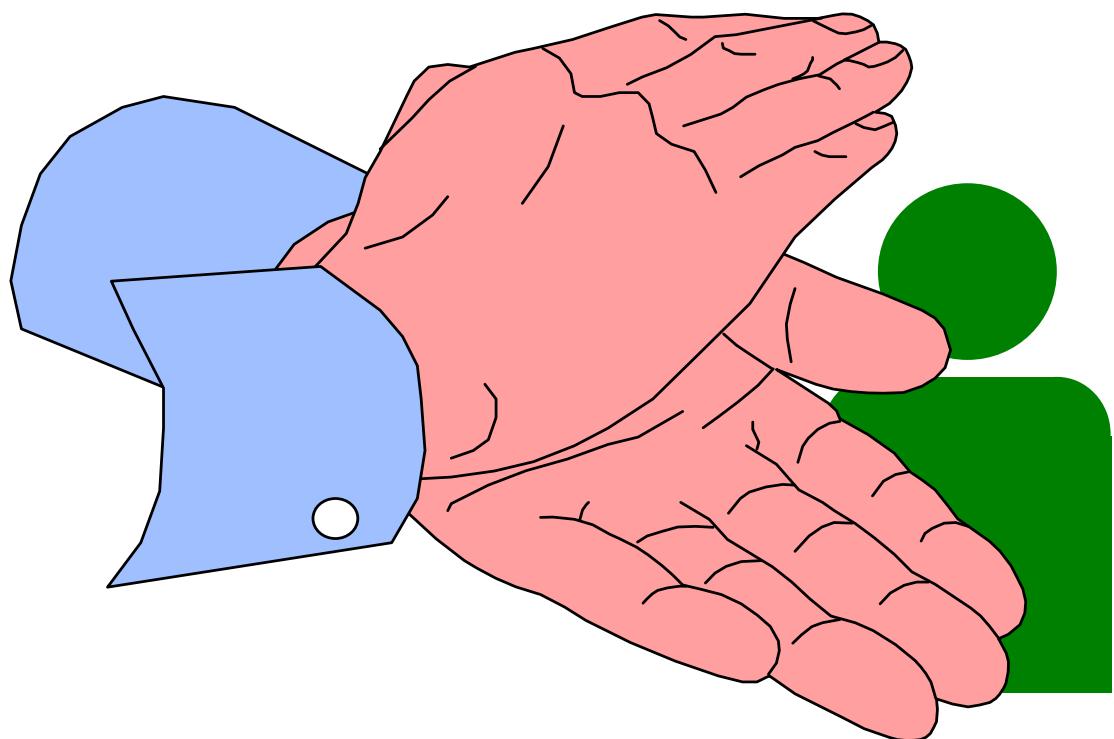


# C++ Fundamentals

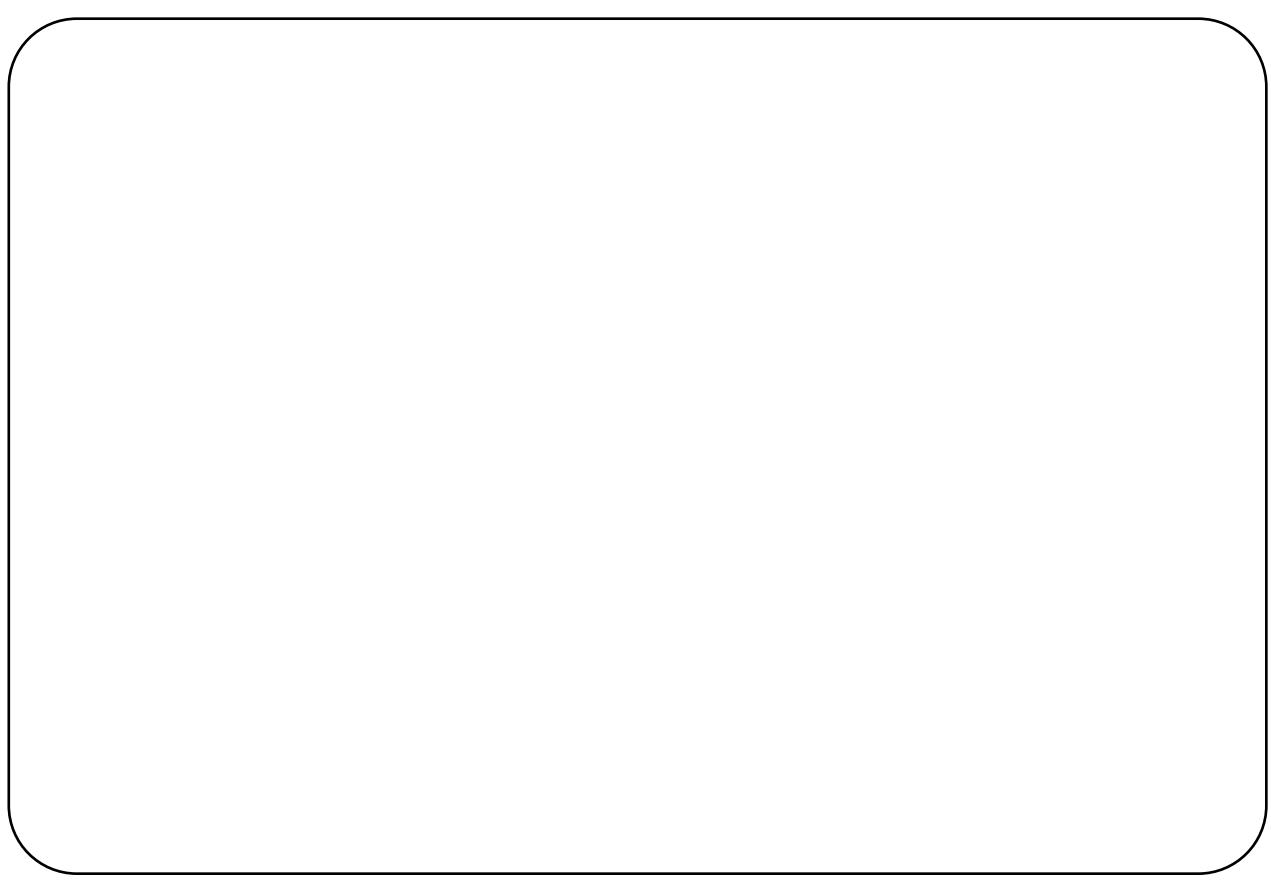


# Chris Seddon

[seddon-software@keme.co.uk](mailto:seddon-software@keme.co.uk)

# C++ Fundamentals

- 1      Introduction to C++
- 2      Data Types
- 3      Expressions
- 4      If Statements and Loops
- 5      Pointers
- 6      Arrays
- 7      Input and Output
- 8      Extensions to C
- 9      Classes
- 10     More on Classes
- 11     Aggregation
- 12     References
- 13     The Preprocessor
- 14     Operator Overloading
- 15     Shallow and Deep Classes
- 16     Further Considerations for Classes
- 17     Standard Template Library
- 18     Inheritance
- 19     Polymorphism
- 20     More Operator Overloading
- 21     Templates
- 22     Exception Handling
- A1     Interrupt Handling
- A2     Real Time Operating Systems
- A3     Optimization
- A4     Real Time Design Patterns



# 1

# Introduction to C++

- **Objective**
  - Get an overview of C++
- **Contents**
  - History
  - What is C++?
  - Compilers
  - Development environments
  - Third party software

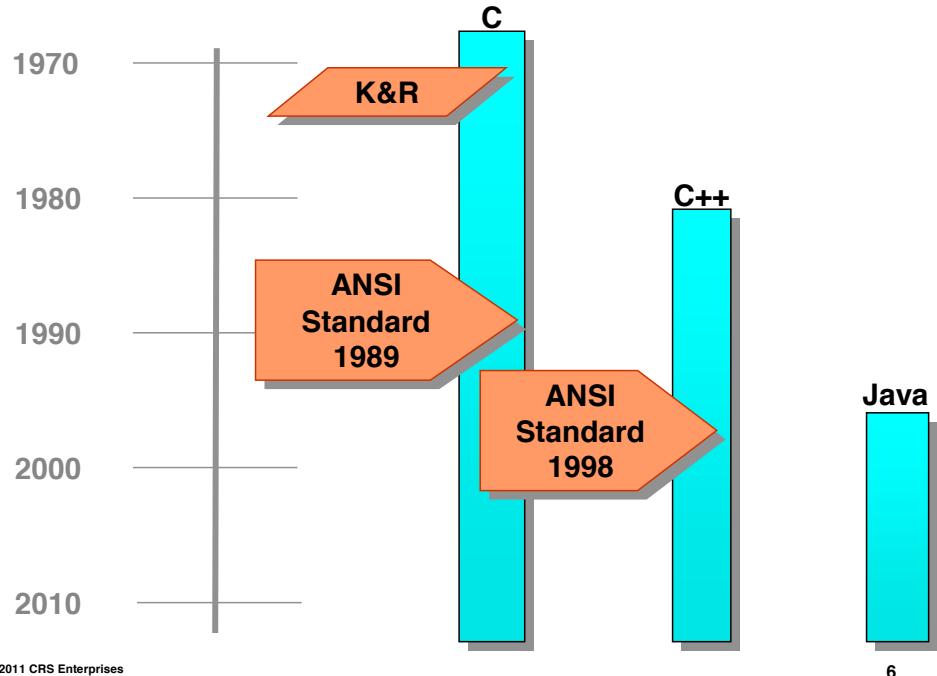


1

C++ is a statically typed, free-form, multi-paradigm, compiled, general-purpose programming language. It is regarded as a "middle-level" language, as it comprises a combination of both high-level and low-level language features. Originally named C with Classes; it was renamed C++ in 1983.

As one of the most popular programming languages ever created,[4][5] C++ is widely used in the software industry. Some of its application domains include systems software, application software, device drivers, embedded software, high-performance server and client applications, and entertainment software such as video games. Several groups provide both free and proprietary C++ compiler software, including the GNU Project, Microsoft, Intel and Borland. C++ has greatly influenced many other popular programming languages, most notably Java.

## History of C/C++/Java

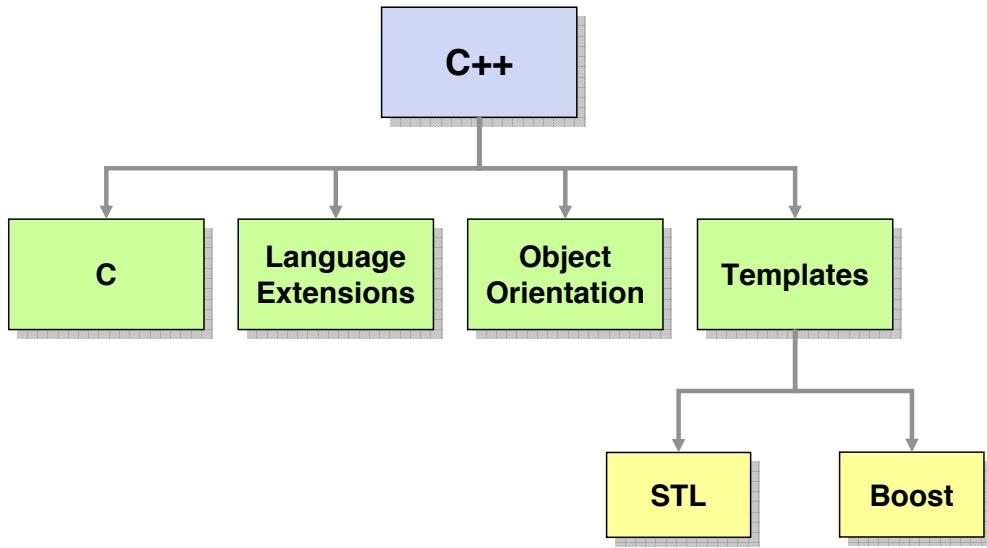


C originated in the years 1969-1973, in parallel with the early development of the Unix operating system. C was written by Dennis Ritchie and Brian Kernighan and underwent many changes in its early years. Finally, in the middle 1980s, the language was officially standardized by the ANSI X3J11 committee, which made further changes. Until the early 1980s, although compilers existed for a variety of machine architectures and operating systems, the language was almost exclusively associated with Unix; more recently, its use has spread much more widely, and today it is among the languages most commonly used throughout the computer industry.

C++ was designed for the UNIX system environment by Bjarne Stroustrup who had studied in the doctoral program at the Computing Laboratory at Cambridge University prior to joining Bell Labs (became AT&T Labs). C++ was not standardized until 1998.

Java has also evolved from C and C++ and is intended as a halfway house between these languages.

## Components of C++



Copyright ©1994-2011 CRS Enterprises

7

C++ is in many ways a evolution of the C language, but with extensive support for Object Orientation. Templates were added in the 1990s and these have transformed the language. Most libraries are written using templates; the Standard Template Library (STL) was shipped as part of the 1998 standard. There have been many extensions to the STL; the Boost library is the modern incarnation of the STL.

## Compilers

- **Microsoft Visual C++**
  - Visual Studio Express 2010
- **GCC**
  - CDT Eclipse Plug-in
- **Other Compilers**
  - Intel, Sun, Comeau, Borland
- **Many Compilers for Embedded Systems**
  - ARM, IAR, Wind River

There are many compilers for C++ which target a huge variety of architectures. Amongst the most popular are Microsoft Visual C++ (incorporated in Visual Studio Express 2008 and 2010) and the Gnu Compiler Collection, GCC. GCC is often bundled with the Eclipse IDE using the CDT Eclipse Plug-in.

Other Compilers include Intel, Sun, Comeau, Borland and there are a multitude of compilers for embedded systems; ARM, IAR, Wind River - to name but a few.

## Standards

- **C++ Standard 1998**
  - included STL
  - highly stable
  - compilers available for wide range of platforms
- **C++ Standard 201x**
  - supposed to be released several years ago
  - originally call C++ 200x
  - **C++ Technical Report 1 (TR1) libraries available now**

C++ was standardized in 1998 and included the STL. It is highly stable and as previously discussed, compilers are available for wide range of platforms.

A new C++ standard is now overdue and is likely to be ratified within 5 years. The standard is often referred to as C++ Standard 201x (originally call C++ 200x) because of this uncertainty. The C++ Technical Report 1 (TR1) libraries are a subset of the new standard and are available now.

## Third Party Software

- **JThreads/C++**
  - high-level thread abstraction library that gives C++ programmers the look and feel of Java threads.
- **Rogue Wave**
  - Extensive set of classes that provide additional functionality to the Standard C++ library
- **CORBA**
  - Language agnostic client, server architecture. CORBA is language agnostic, but has bindings for many languages including C, C++, Java and Python

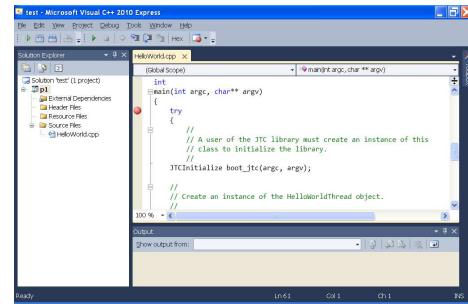
JThreads/C++ is the short-form of "Java-like Threads for C++". JThreads/C++ is a high-level thread abstraction library that gives C++ programmers the look & feel of Java threads.

Rogue Wave is an extensive set of classes that provide additional functionality to the Standard C++ library.

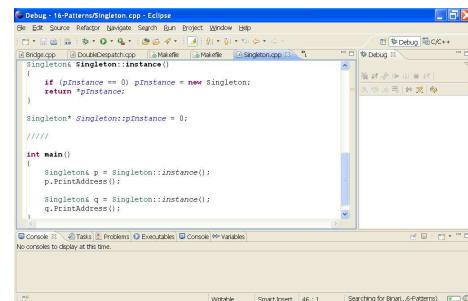
The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) that enables software components written in multiple computer languages and running on multiple computers to work together (i.e., it supports multiple platforms). CORBA is language agnostic, but has bindings for many languages including C, C++, Java and Python

# Development Environments

- Microsoft Developer Studio
  - old code base
  - excellent compiler
  - only Windows



- Eclipse with CDT plug-in
  - mature
  - younger code base
  - cross platform



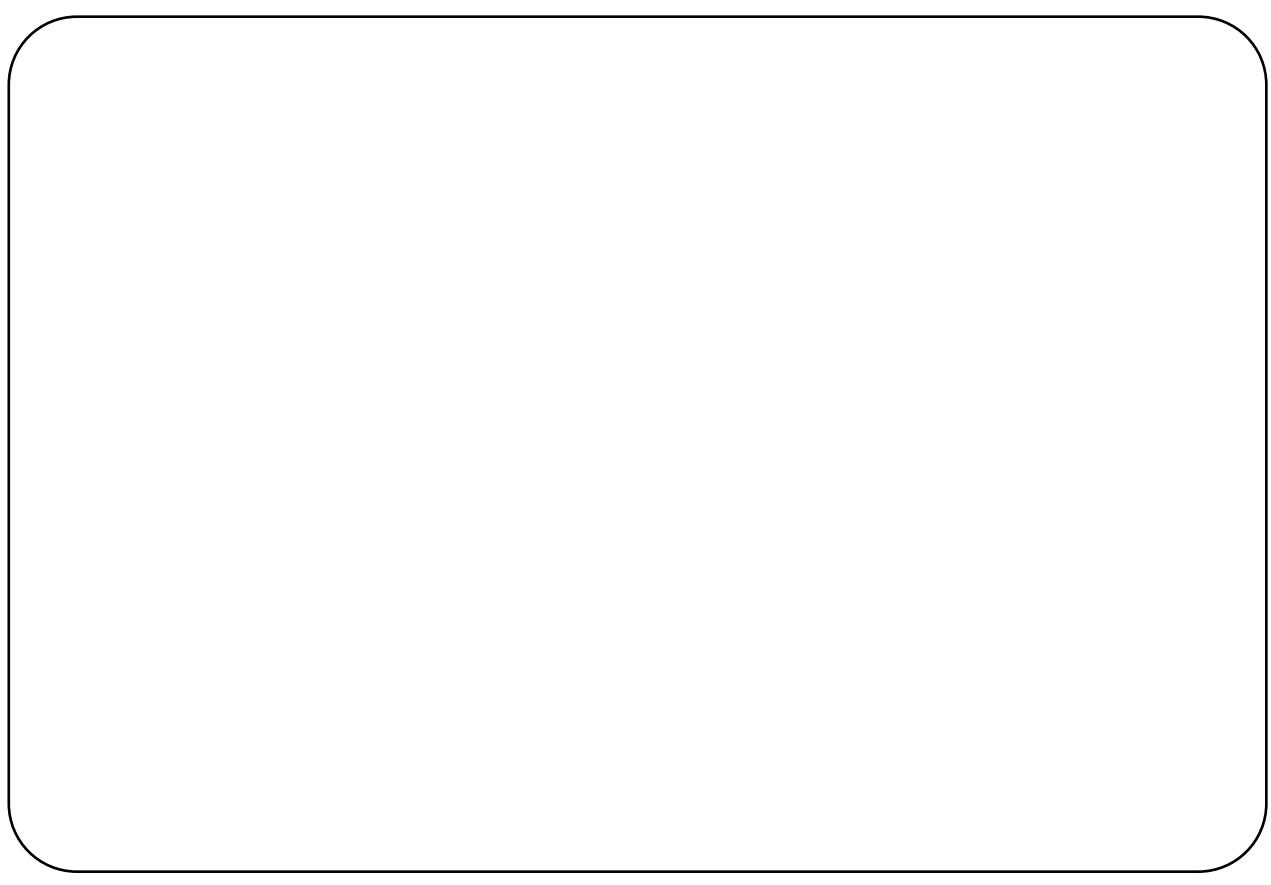
Copyright ©1994-2011 CRS Enterprises

11

Traditionally C++ developers have shunned development environments, preferring to use emacs and vi editors. However in recent year many programmers have started using IDEs.

The most popular IDEs are Microsoft Developer Studio and Eclipse. These two environments are broadly similar, thought the Microsoft code base is now 20 years old (making it difficult to change).

Using Microsoft Developer Studio restricts development to Windows, but Eclipse is written in Java and can be used everywhere.





# 2

## Data Types

- **Objective**
  - Explore C++'s built-in scalar types
- **Contents**
  - Variable declaration format
  - Rules for legal identifiers
  - Scalar types
  - Initializing variables
  - Constants



# 2

Copyright ©1994-2011 CRS Enterprises

15

C++ defines four classes of built in types: integer, floating, bool and character.

Several integral types exist including short, int and long. Each of these types map to a specific number of bytes on any target machine, but everything is platform dependent. For example on 32 bit Intel platforms running Windows an int is defined as 32 bits, but 64 bit Intel running Linux will have 64 bit ints. This means C++ programs are not easily ported between 32 and 64 bit platforms. Subtle problems can also occur in porting between similar platforms.

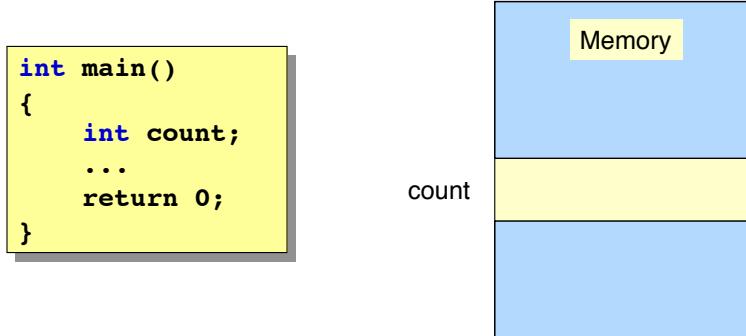
Single and double precision floating point are available (you can even use 80 bit IEEE floating point by declaring long double). Single precision (float) is only accurate to 6 significant figures, whereas double precision (double) is accurate to 14 significant figures

Boolean types exist and are straightforward. Bear in mind these types were not added to the language until just before C++ was standardized. Older programs often simulate Booleans as integers.

Finally support for characters and strings is provided, though complicated. Originally only ASCII characters were available (char), but nowadays there is full support for Unicode. Strings were originally implemented as character arrays, but the Standard library now has full support for string classes. Using string classes is more straightforward than using character arrays.

## Variable Declarations

- A variable declaration reserves an area of memory and gives it a symbolic name
- The type determines the amount of memory reserved and the values that it can contain



Copyright ©1994-2011 CRS Enterprises

16

All data must be declared before it can be used. This is achieved using a variable declaration; the declaration reserves an area of memory and gives it a symbolic name. Every variable must be associated with a type; the type determines the amount of memory reserved and the values that it can contain. Variables cannot change type during the execution of a program - C++ is a strongly typed language.

## General Declaration Format

- A declaration consists of
  - a type
  - a comma separated list of variables of that type

```
int main()
{
    int    highval,lowval;
    float  celcius,fahr,maxtemp;
    char   temp_abbreviation;
    int    count;
    count = 5;
    double earth_radius; ←
    ...
    ...
    return 0;
}
```

*data must be declared before use*

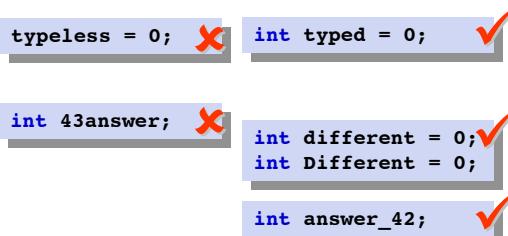
*declarations can be placed after code*

You can declare several variables for a given type all at once - use a comma separated list. In C all variables must be declared at the top of a block, before first use. This is not true in C++ where variables can be declared anywhere in a block (but before first use). It is common practice to declare a variable as close to the code using it.

# Identifiers

- **C++ identifiers**

- declare before use
- CaSe sensitive
- letters, digits, underscore
- start with a letter or underscore



- **C++ keywords**

- reserved
- cannot be used as identifiers

`int double ...`

*underscore is considered a letter*

C++ uses identifiers for variables and function names. Each identifier is case sensitive and consists of letters, digits, underscore. A identifier must start with a letter or underscore, not with a digit. The C++ compiler often uses a double underscore for its own identifiers, so try to avoid identifiers starting with two underscores. Many programmers use a leading or trailing underscore in identifiers.

C++ keywords are reserved and cannot be used as identifiers. If you are not sure which identifiers are reserved, don't worry, the compiler will soon let you know!

# Fundamental Types

- **character types**
  - **char (8 bit Ascii)**
  - **wchar\_t (16 bit Unicode)**
- **integral types**
  - **short, int, long**
  - **signed, unsigned**
  - **usually 32/64 bit**
- **boolean**
  - **true and false**
- **decimal types**
  - **float, double, long double**
  - **float has 7 sig. digits**
  - **double has 13 sig. digits**

```
char    c1 = 'a';
wchar_t c2 = L'a';

int    x1 = 5000;
short  x2 = 2
long   x3 = 6712354L;

bool flag = true;

float  n1 = 45.7134F;
double n2 = 681.7312;
```

C++ has a limited number of built in types: they are classified as character, integral, boolean and decimal types. In practice this does not turn out to be a problem - C++ uses classes to define your own types.

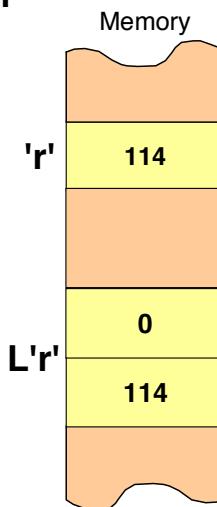
Character types were originally restricted to using the Latin character set and hence were defined as 8 bit characters. Nowadays, internationalisation means we must cater for many character sets such as those used in Arabic, Indian and Chinese scripts. C++ allows the wide character type (Unicode) to define characters in these scripts.

Several integral types are defined; the most important of which is int. This type nearly always has the same size as the underlying word size of the OS. Thus an int is 32 on most operating systems, but with the advent of 64 bit Windows and Unix this is set to change. To save space you can use short or to gain precision you can use long. Both signed and unsigned types are supported.

Decimal types also come in a variety of types. In modern applications you should avoid float because of its lack of precision; double suffices for most applications. The long double type is now obsolete.

## Character Types

- **char** is used to store a single character
  - its value is invariably the ASCII code
  - use single quotes
  - usually 1 byte
- **wchar\_t** is for Unicode
  - 16 bit characters
  - Arabic, Greek, Japanese, Chinese ...
  - Unicode Standard



char is used to store a single character and although a compiler can support any internal character set, nowadays its value is invariably an ASCII code. Characters are defined using single quotes; double quotes are reserved for strings (character arrays). Virtually every compiler uses 1 byte per character.

wchar\_t is used for Unicode (16 bit) characters. Arabic, Greek, Japanese, Chinese and many other character sets are supported using the Unicode Standard (ISBN 0-321-18578-1). To distinguish a Unicode character from a normal character use the prefix L. Note that the first 256 characters in Unicode are the Latin characters (hence the leading zero in the definition of Unicode L'r').

## Example

```
#include <stdio.h>

int main()
{
    char x;
    char nl;

    x = 'a';
    nl = '\n';
    printf("x = %c\n", x);
    printf("x = %i\n", x);
    printf("x + 1 = %c\n", x + 1);
    printf("New%cline\n", nl);
    return 0;
}
```

%c for char

%i for ASCII code

(x + 1) will be 'b'

This example uses printf to output a character variable - use %c in the format string. Note that characters can in some sense be considered as integers. If you print a character using %i you will reveal its underlying ASCII representation. Not only that, but you can add one to a character to get the next character in the set. In ASCII this means an 'A' will become a 'B'; a 'B' will become a 'C' and so on.

# Integral Types

- **Types**
  - **short, int, long, signed and unsigned**
- **Storage**
  - short int <= int**
  - int <= long int**
  - short must be at least 16 bits long**
  - long must be at least 32 bits long**
- **C99 has additional types**
  - **fixed size ints**
  - **booleans**
  - **complex number**

```

short      s1 = 2;
short int s2 = 2;

int        i = 5000;
unsigned int i = 5000;
signed int  i = 5000;

long     x3 = 6712354L;
long     x3 = 6712354L;

```

Integral types follow the following rules:

**short int <= int**  
**int <= long int**  
**short must be at least 16 bits long**  
**long must be at least 32 bits long**

Note that the actual storage size of these types is not defined by these rules and are compiler dependent.

Short ints can be abbreviated to:

**short**

Likewise you can abbreviate long int to

**long**

Typically, int and long are 32 bits long on a 32 bit machine (64 bits on a 64 bit machine). short ints are usually half the size of ints.

## Floating Point Numbers

- Floating point types hold fractional and large numbers
  - *float* holds a single precision floating point number
  - *double* holds a double precision floating point number
  - *long double* provides even greater accuracy and precision
- Modern code uses *double*
  - use postfix F or f to define a literal constant if single precision is necessary

```
int main()
{
    double jeopardy = 3.75;
    float away = 3.75F;
    ...
    return 0;
}
```

Copyright ©1994-2011 CRS Enterprises

The diagram shows a vertical list of floating-point literals. To the right of each literal, an arrow points to its scientific notation equivalent. The literals are:

- 3.14159265359; →  $3 \times 10^8$
- 273.15
- 4.0
- 0.0
- 273.
- .15
- 17.5F
- 299792.458
- 3e8f ←  $3 \times 10^8$
- 6.672E-11F ←  $6.672 \times 10^{-11}$

23

Use floating point types hold fractional and large numbers. Avoid using float because it has limited precision and can lead to significant rounding errors. Most programs use double for all floating point numbers, but long double is available to provide even greater accuracy and precision if required.

## Exercise

- Which names are legal in the code fragment below?

```
int main()
{
    char  code;
    int   employee_number, 999number, age-in-1986;
    int   EMPLOYEE_DEPT, emp_count;
    float _salary, overtimeHours;
    char  cEmpCode;
    int   double;
    ...
    return 0;
}
```

Most of the identifiers above are legal. The identifiers that you can't use are

<b>999number</b>	- begins with a digit
<b>age-in-1986</b>	- hyphen is not permitted
<b>double</b>	- double is a keyword

Note that EMPLOYEE\_DEPT is allowed, but discouraged; identifiers are usually in lower case with subsequent words capitalised (overtimeHours) or separated by underscores (emp\_count). Note that Microsoft (but none else!) recommend Hungarian notation, where the leading characters of the identifier indicate its type (cEmpCode). This was always a poor convention and is now falling into disuse.

## Initializing Variables

- **Variables may be initialized to specific values in their declarations**
- **Global variables default to zero**
- **Local variables not initialized automatically**

```
int main()
{
    int      i = 0;
    int      mins_in_day = 24 * 60;
    double   base_rate = 3.75;
    char     letter_b = 'b';
    ...
    return 0;
}
```

Variables may be initialized to specific values in their declarations. Because of the way storage is allocated by C++ compilers, you will find that all initialized global variables will default to zero, but local variables will not be initialized automatically - they will have an arbitrary value (whatever happens to be in memory when their storage is allocated).

## Constants

- Constants must be initialized at declaration time
  - r-value
  - compiler forbids assignment

```
int main()
{
    const int    MONTHS = 12;
    const double PI = 3.14159;
    const char   SPACE = ' ';
    ...
    return 0;
}
```

- Older style uses the preprocessor

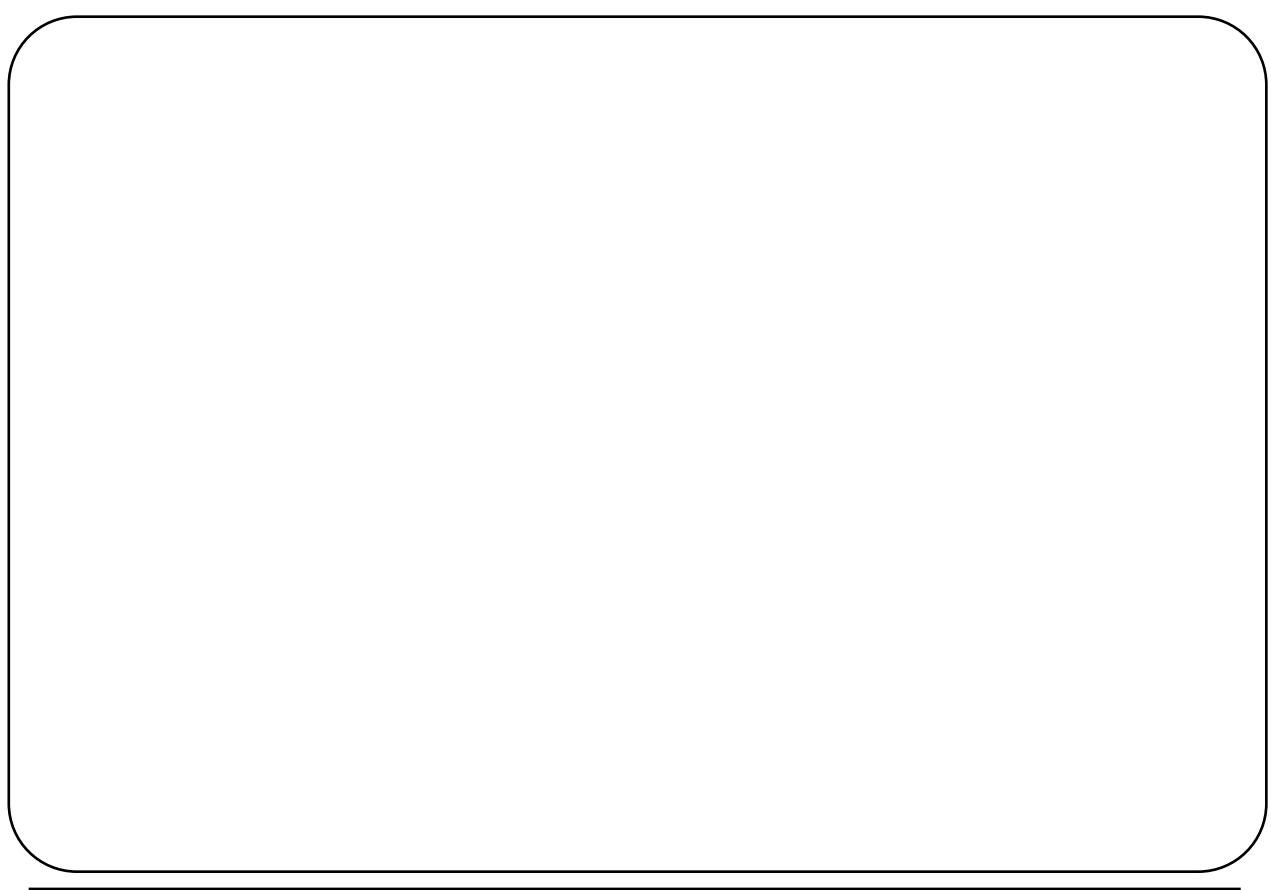
```
#define PI = 3.14159
```

*no semicolon*

Constants can be defined in C++ programs using the `const` qualifier. The compiler will ensure constants are not modified after definition, by only permitting the use of a constant in the right hand side of an assignment (hence the term r-value). Variables can be placed on the left hand side of an assignment (hence the term l-value).

A common convention is to use block capitals for constants - to make them stand out.

Early compilers did not use the `const` keyword and constants had to be defined in the pre-processor. Note that if you use the pre-processor to define a constant do not terminate the definition with a semi-colon (the pre-processor is not C++!). The pre-processor will be discussed later in the course.



# 3

# Expressions

- **Basic Operators**
  - Binary operators
  - Logical operators
  - Bitwise operators
- **Other Operators**
  - Side effect operators
  - Cast operators
- **Integer Arithmetic**
- **Booleans**



# 3

Copyright ©1994-2011 CRS Enterprises

29

In this chapter we investigate C++'s basic operators. These include binary, logical and bitwise operators. We also discuss C++ side effect operators and cast operators.

The chapter concludes with a discussion of integer arithmetic and support for booleans.

# Operators

- **Binary**

**+** add

- **-** subtract
- **\*** multiply
- **/** divide
- **%** remainder

```
x + y;
x - y;
x * y;
47 / 10;
47 % 10;
```

The diagram shows the division of 47 by 10. The quotient is 4 and the remainder is 7. Arrows point from the numbers 47, 10, and the result 4 to their respective parts in the code.

- **Logical**

**&&** AND

**||** OR

**!** NOT

```
if(x > 3 && y < 7) ...
if(!(x <= 3) || (y >= 7) ...
```

- **Bitwise**

**&** AND

**|** OR

The division operator behave differently for integral and floating types. For floating types, division is exact (at least to the precision of the underlying type). However, if both data items are integral, then integer division is performed and the remainder is discarded.

If the remainder is important then it can be retrieve using the **%** operator. The remainder operator can only be used with integral operands.

There are two types of AND operation. Logical **&&** is the Boolean AND; Bitwise **&&** works at the binary level and is used for low level operations. Similarly considerations apply for **||** and the NOT operators.

# Assignment

```

int main()
{
    int a = 10, b = 20, c = 30, d = 40;

    a++;
    a += 1;
    a = a + 1;

    d = d - b;
    d -= b;

    c = c * (b + 10);
    c *= b + 10;

    return 0;
}

```

Copyright ©1994-2011 CRS Enterprises

31

Rather confusingly, there are 3 different ways to add one to a variable. Historically, these operators generated different code, but with modern optimizing compilers this is no longer the case. Choose the form that you consider the most readable.

The increment and decrement operators are defined as

**++ increment by 1**  
**-- decrement by 1**

Other operators are as follows:

<b>+= update by addition</b>	
<b>-= update by subtraction</b>	
<b>*= update by multiplication</b>	
<b>/= update by division</b>	
<b>%= update by modulus (ints only)</b>	
<b> = update by BIT-ORing</b>	
<b>&amp;= update by BIT-ANDing</b>	
<b>^= update by BIT-XORing</b>	
<b>&lt;&lt;= update by BIT SHIFTing left</b>	
<b>&gt;&gt;= update by BIT SHIFTing right</b>	

## Side Effects of ++ and --

- Prefix and Postfix Operators

– increment is a side effect

```
#include <stdio.h>
int main()
{
    int i, x;
    i = 100;
    x = ++i;
```

```
#include <stdio.h>
int main()
{
    int i,x;
    i = 100;
    x = i++;
```

**= i++      assignment first, then the side effect**

**= ++i      side effect first, then the assignment**

The prefix and postfix operators are called side effect operators. The differences between the operators only becomes apparent when the operators are used within an expression.

If these operators are used in prefix mode then the effect of the operator is immediate; the operand is updated before any assignment.

In postfix mode the reverse is true; the operand is updated after the assignment.

Note:

The order of evaluation of multiple side effect operators on the same operand is not defined by the C++ standard. This can give rise to ambiguous code.

## Exercise 1 - using ++ and --

- What value is the value of a after each line executes?

```
#include <stdio.h>

int main()
{
    int a = 0, b = 10, c = 1;

    a = ++b + ++c;
    a = b++ + c++;
    a = ++b + c++;
    a = b-- + --c;
}
```

Try this out in a debugger to check your results.

## Casting

- **Type conversions (casts) allow you to assign different types at run time**
  - with possible loss of precision

```
#include <math.h>
int main()
{
    double pi = 4 * atan(1.0);
    float x;
    int i;

    x = (float) pi;
    i = (int) pi;
}
```

Copyright ©1994-2011 CRS Enterprises

34

Casting is where data of one type is being assigned to a variable of another type. The compiler will allow casts between compatible types. Note that casting may lose precision (e.g. double to float).

Note:

The programmer has some control when dealing with constants. The letters L and F can be used to indicate the desire to treat the constant as a long (or long double) or a float respectively.

## Arithmetic Involving Different Types

- **Compiler will promote types (automatic cast) where appropriate**
  - but only in a sub expressions
  - watch out for truncation in integer division
  - watch out for loss of precision in assignment

```
int main()
{
    int     x = 10;
    double y = 7.77;
    int     result;
    ...
    result = 15 / x + y;
}
```

The compiler will perform automatic casts in sub-expressions involving different types. The automatic conversion is not sensitive to the context and depends only on the operands in the sub-expression.

This leads to unexpected results in expressions such as

**15 / x + y;**

This is evaluated as two sub-expressions. In the first sub-expression

**15 / x**

both operands are integral and the compiler does not cast either operand. This leads to integer division and a result of 1.

The second sub-expression is then

**1 + y**

and this time the operands are of different types. The compiler will promote the first operand to the double 1.0000. Obviously the decimal part has already been discarded and the final result is 8.77 and not 9.27 as you might have first thought.

## Relational Operators

- **Binary operators**

**>** greater than  
**<** less than  
**==** equal to  
**!=** not equal  
**>=** greater than or equal  
**<=** less than or equal

```
int main()
{
    int    x = 100;
    int    y = 200;

    if(x == y) {
        ...
    }

    if(x != y) {
        ...
    }
}
```

All six relational operators exist in C++. Care must be taken with the syntax of the equality and non-equality symbols, since neither **!=** nor **==** are common in other languages.

Operands should all be of integer or compatible type. Automatic conversions will take place on non integral types. The Boolean value generated by these operators are either the integer 1 (true) or the integer 0 (false).

Be careful to avoid comparison involving floating point numbers, as these are not represented exactly in memory and small rounding errors can cause unexpected results (especially **==**).

## Booleans

- C did not have a Boolean type
  - integers used instead
  - Zero is false
  - Non-zero is true
- C++ does have a boolean type
  - true, false
- You can also use #define
  - to define your own Booleans

```
int main()
{
    int number = 147;
    int result;
    bool flag = false;

    result = !number;
    result = !result;

    return 0;
}
```

```
#define TRUE 1
#define FALSE 0
```

C didn't have a Boolean type, so integers were used for that purpose. C++ does have a Boolean type, but old habits die hard - you are very likely to encounter the old style.

Any non-zero integer is treated as TRUE by the compiler (e.g. number = 147); zero is treated as FALSE. The relational operators are guaranteed to yield 0 or 1, but are capable of interpreting ALL integral values as Booleans.

In attempt to make C++ code more readable, many programmers define their own Booleans using the preprocessor:

```
#define TRUE 1
#define FALSE 0
```

## && Operator

- **logical AND operator**

```
false && false = false
false && true  = false
true  && false = false
true  && true  = true
```

- **short-circuit evaluation**

- expressions evaluated from left to right
- evaluation stops as soon as result is known

*divide by zero doesn't happen*

```
int main()
{
    int a = 42;
    int b = 0;
    int result;

    if(a > 50 && 15/b > 1) {
        ...
    }
}
```

Unlike the majority of C++ operators, the AND and OR operators are evaluated in strict order. Both perform evaluation from left to right. If the truth value of the expression is determined on performing the left operand evaluation, further evaluation does not ensue. This is called short-circuit evaluation and is important in conditions like

**a > b && 15/(a -b)**

where the second expression could potentially cause a divide by zero error. The short circuit evaluation ensures that this never happens, because if a - b does equal zero, the first condition will fail and the short circuit will return FALSE immediately.

Care must be taken to use the repeated symbols **&&** and **||**. The symbols **&** and **|** on their own are used for bit manipulation.

## II - the OR Operator

- **logical OR operator**

```
false || false = false
false || true  = true
true  || false = true
true  || true  = true
```

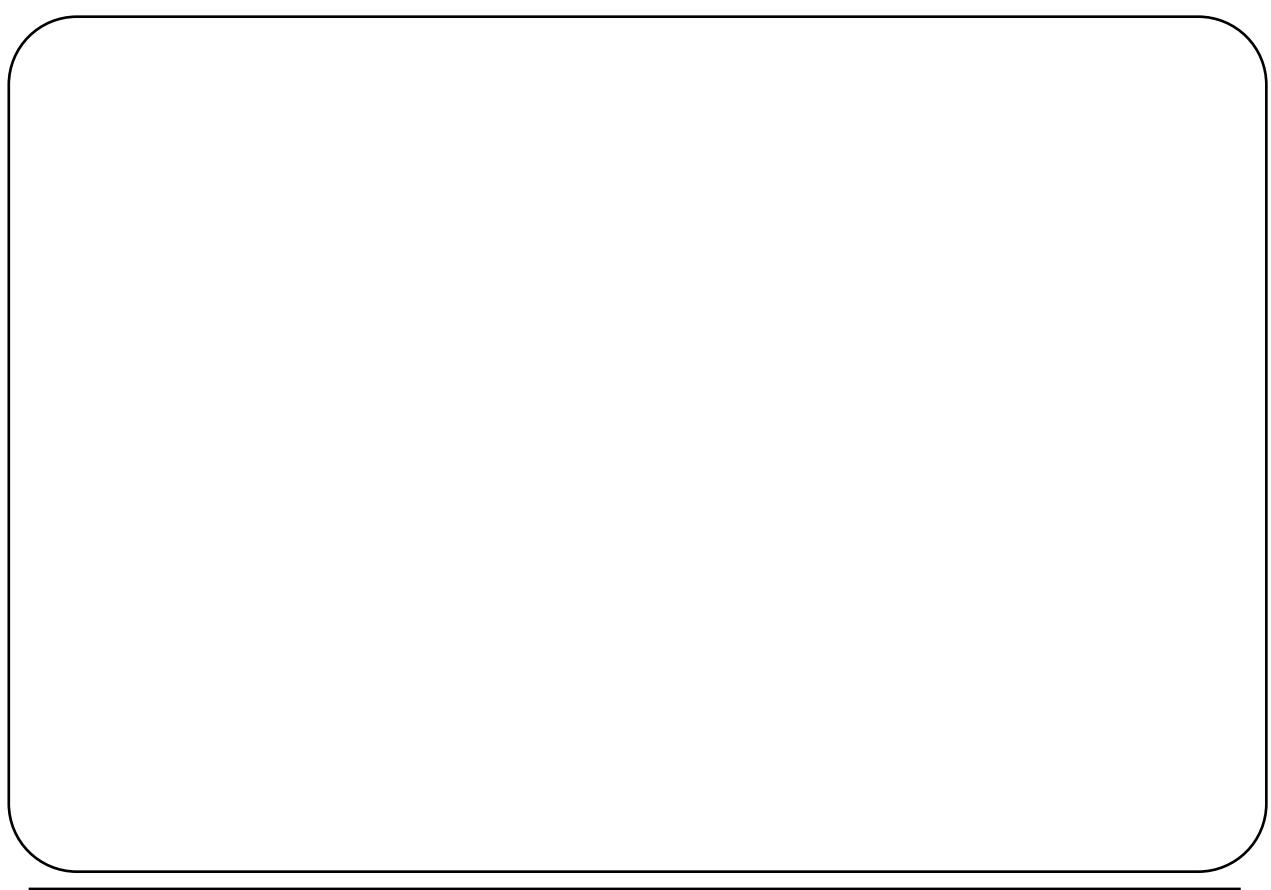
- ***short-circuit* evaluation**

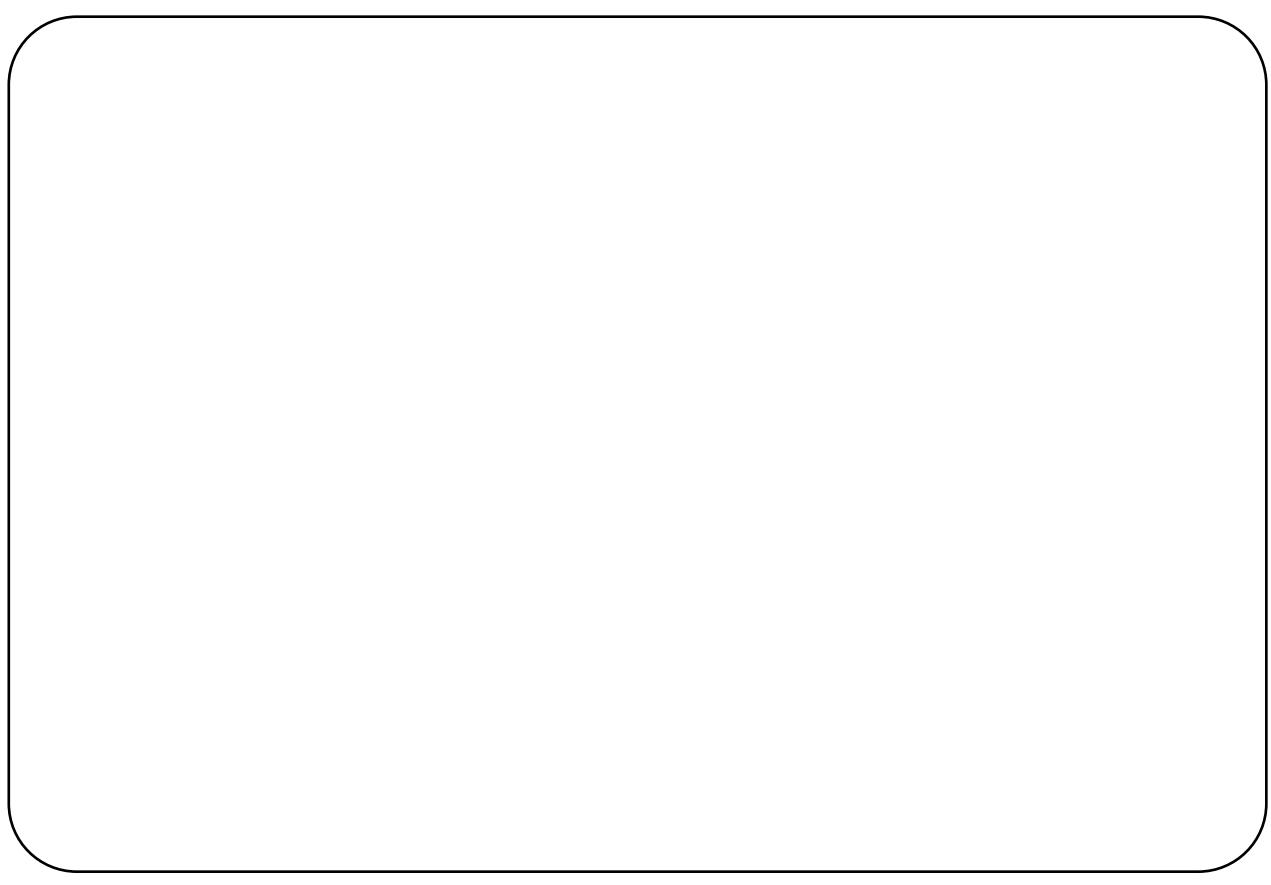
– as with &&

```
int main()
{
    int x = 0;
    int y = 22;

    if(x > 10 || y < 30) {
        ...
    }
}
```

Similar considerations apply to the OR operator.





# 4

## If Statements and Loops

- **Conditionals**
  - **if statement**
  - **switch statement**
  - **conditional operator**
- **Iteration**
  - **while statement**
  - **do while statement**
  - **for statement**



4

Copyright ©1994-2011 CRS Enterprises

43

C++ inherits its conditional and iteration (loop) statements from C.

There are three types of conditional:

**if statement**  
**switch statement**  
**conditional operator**

And there are three types of loop:

**while statement**  
**do while statement**  
**for statement**

## ***if Statement***

```
if (expression)
    statement
```

```
if (d > 0.0) printf("d is positive\n");
```

```
if (d > 0.0)
{
    printf("d is positive");
    printf("\n");
}
```

The simplest form of the if statement is shown above. The conditional expression must be enclosed with parentheses and is followed by the body. The body can be either a single statement or a block of statements.

## if - else Statement

```
if (expression)
    statement
else
    statement
```

```
if (d > 0.0)
    printf("d is positive\n");
else
    printf("d is not positive\n");
```

```
if (d > 0.0){
    printf("d is positive");
    printf("\n");
} else {
    printf("d is not positive");
    printf("\n");
}
```

If statements can have an else branch. The body of the else branch can be a single statement or a block of statements. It is perfectly permissible to mix a single statement in the if branch with a block in the else branch or vice-versa.

Thus there are 4 forms of the if-else statement:

- if(condition) statement; else statement;**
- if(condition) statement; else block-of-statements;**
- if(condition) block-of-statements; else statement;**
- if(condition) block-of-statements; else block-of-statements;**

## Multiple *if* Statements

```
if (expression)
    statement
else if (expression)
    statement
...
else
    statement
```

```
#include <stdio.h>

int main()
{
    int legs = 6;

    if (legs == 1)
        printf("Flamingo\n");
    else if (legs == 2)
        printf("Ape\n");
    else if (legs == 4)
        printf("Dog\n");
    else
        printf("Insect\n");
}
```

- This form allows for ‘Multi-way decision making’

By combining several if statements you can create multi-way decision logic. Note that this is not new syntax; it is simply multiple application of the if statement. If you choose the layout shown above, the multi-way decision becomes very readable.

## The *switch* Statement

- **Multiway selections**
  - expression determines case
  - break to exit the switch
  - drop through if break isn't used
  
- **Case index must be integral**
  - or char
  - can't use strings for cases

```
switch(legs)
{
case 1:
    printf("Flamingo\n");
    break;
case 2:
    printf("Ape\n");
    break;
case 4:
    printf("Dog\n");
    break;
default:
    printf("Insect\n");
    break;
}
```

Tests based on the evaluation of a simple integral expression can be expressed using the switch statement. The switch is neater and more efficient than a corresponding nested if/else construct. Once the syntax is in place, the flow is relatively easy to read. The rules are few but important and are detailed on the next page.

Each case can contain multiple statements and it is not necessary to enclose these statements in {}. The break statement transfers control to the end of the switch. Without the break, control simply passes to the next case, and although perfectly legal, probably not what was intended (but see next page for a valid example of omitting the break).

Note that the switch only works with integral types.

## No *break* in *switch* Statement

```

int verse ;
for (verse = 1; verse != 5; ++verse)
{
    printf("On the ");
    switch (verse)
    {
        case 1 : printf("1st"); break;
        case 2 : printf("2nd"); break;
        case 3 : printf("3rd"); break;
        default : printf("%dth", verse); break;
    }
    printf(" day of Christmas my true love sent to me:\n");
    switch (verse)
    {
        case 5 : printf("five gold rings, ");
        case 4 : printf("four calling birds, ");
        case 3 : printf("three french hens, ");
        case 2 : printf("two turtle doves, and ");
        case 1 : printf("a partridge in a pear tree\n");
    }
}

```

Copyright ©1994-2011 CRS Enterprises

48

In this program the *break* statements have been omitted in the second *switch* statement to give a drop through effect. As you can readily see, this can be useful occasionally. A more common usage for the drop through would be:

```

case 3:
case 4:
case 5;
// statements for all three cases

```

## The ?: Conditional Operator

- 3 operands
  - expression1 ? expression2 : expression3

```
max = (x > y) ? x : y;  
min = (x < y) ? x : y;
```

- More concise than *if* statement
  - but more cryptic?

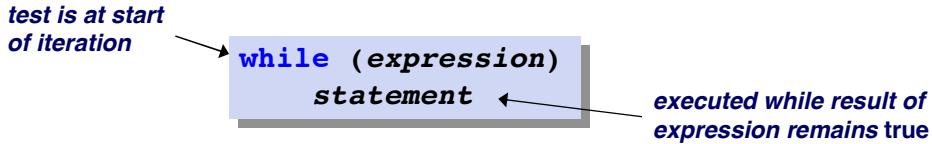
```
if (x > y)  
    max = x;  
else  
    max = y;
```

The ternary conditional expression operator is an efficient alternative to a simple if/else construct. Care must be taken to ensure type matching and precedence.

The conditional expression operator was used extensively to optimize code before the advent of optimizing compilers - nowadays its use is a style choice.

## The **while** Statement

- The simplest iterative statement



```
#include <stdio.h>

int main()
{
    int x = 0;
    while (x < 5)
        printf("%d\n", x++);
    return 0;
}
```

Copyright ©1994-2011 CRS Enterprises

50

The while loop consists of a test on entry, followed by a statement. This statement can be a simple statement (as shown above) or a block of statements enclosed in {}. The statement/block is executed only if the test evaluates to true. Note that if the test fails on the first iteration, the block will never be executed.

## Compound Statement

- The statement part of the loop can be a simple statement or a compound statement

```
#include<stdio.h>

int main()
{
    int x = 0;
    while (x < 5)
    {
        printf("%i\n", x);
        x++;
    }
    return 0;
}
```

*compound Statement*

The example above shows how to use a block as the body of loop. The block is called a compound statement. Normally a block consists of two or more statements, although it is legal to place a single statement (or even no statements) in a block.

## The *do while* Statement

- Test is at the end of loop



```
#include <stdio.h>

int main()
{
    int x = 0;
    do
        printf("%d\n", x++);
    while (x < 5);
}
```

Copyright ©1994-2011 CRS Enterprises

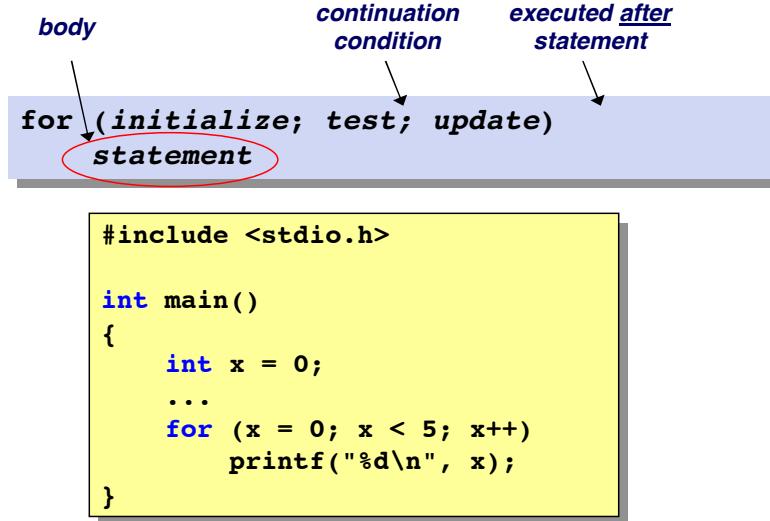
52

The do while loop is the inverse of the while statement in the sense that the test is at the end of the body. The body is repeatedly executed while the test evaluates to true. This form of loop implies the body is executed at least once.

Note the syntax. The do and the while enclose a block of statements. As with the switch statement, the block does not have to be enclosed in {}. A common error is to forget to enclose the test in parentheses.

## The **for** Statement

- The generic iterative statement



Copyright ©1994-2011 CRS Enterprises

53

The for loop is an alternative to the while statement, with separate clauses for initialization, test, and update. The body can be a single statement or block of statements. Note that the initialization, test, and update clauses are separated by semi-colons.

The order of the three clauses and body is as follows:

**initialize**

is performed once only and then

**test**

**body**

**update**

are repeated until the test fails.

Note that the initialise and update clauses can be blank or can contain multiple comma separated expressions:

`for( ; x < 5; x++) { ... }`

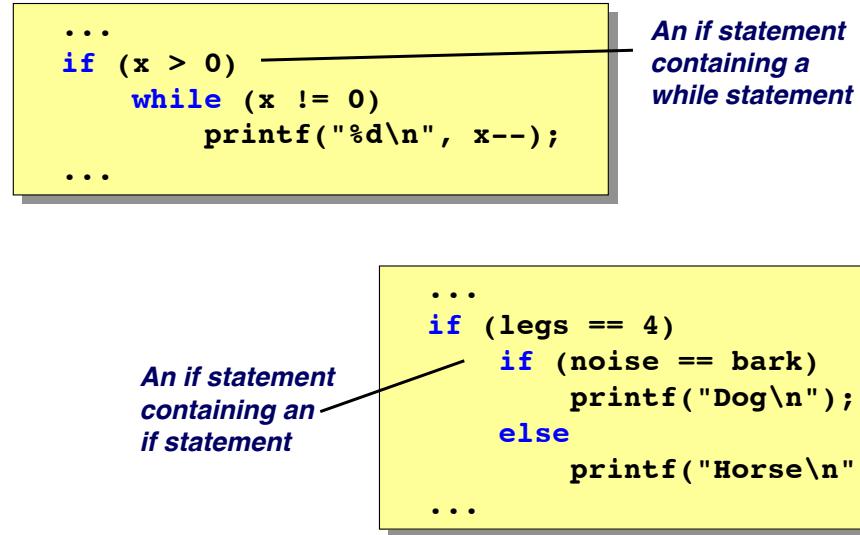
`for(a = 0, b = 0; a + b < 5; a++, b++) { ... }`

The test can also be omitted, in which case we get an infinite loop. In such cases a break statement in the body such as

`if(x >= 5) break;`

can be used to exit from the loop. This form of the break statement should not be confused with the break statement used in switch. Same syntax, but different semantics!

## Nested Control Flow



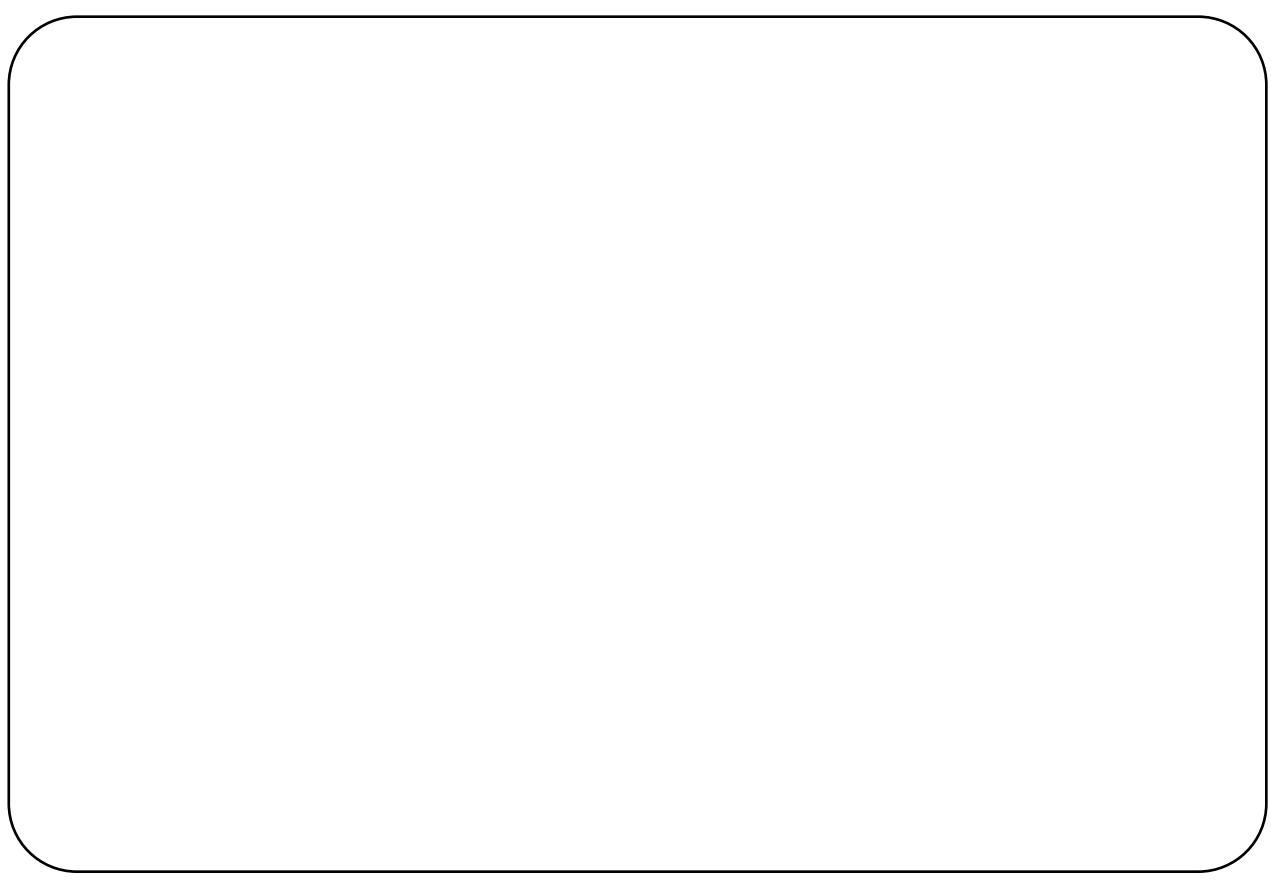
The body of an if or while can be a simple statement. This statement can itself be an if or while statement. If this statement spans several lines readability suffers. Many coding styles insist on the presence of braces for all these constructs to make the extent of the body obvious, even when there is only one statement in the body. The first example could be rewritten as:

```

if (x > 0)
{
    while (x != 0)
    {
        printf("%d\n", x--);
    }
}

```

The other problem is more serious. Without the presence of braces, the procedural meaning can be changed. The rule is: An "else" will be paired off with the nearest "if" which does not have an "else".



# 5

# Pointers

- **Pointers**
  - the concept
  - uses of pointers
  - declaring pointers
  
- **Operators**
  - & operator
  - \* operator
  
- **Complex Pointers**
  - pointers to pointers



**5**

Copyright ©1994-2011 CRS Enterprises

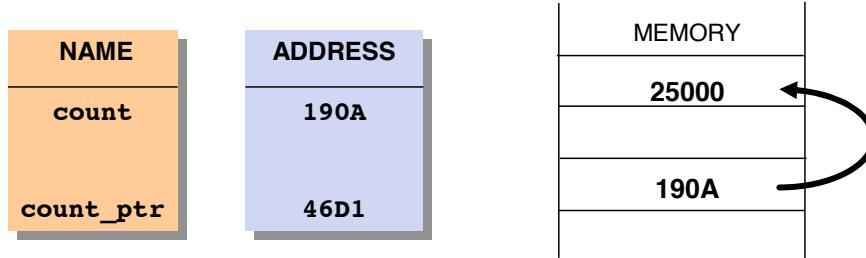
57

A pointer is a programming language data type whose value refers directly to (or "points to") another value stored elsewhere in the computer memory using its address. For high-level programming languages, pointers effectively take the place of general purpose registers in low-level languages such as assembly language or machine code, but may be in available memory. A pointer references a location in memory, and obtaining the value at the location a pointer refers to is known as dereferencing the pointer.

Pointers to data significantly improve performance for repetitive operations such as traversing strings, lookup tables, control tables and tree structures. In particular, it is often much cheaper in time and space to copy and dereference pointers than it is to copy and access the data to which the pointers point.

## Indirection

- A pointer is a variable that contains the address of another variable
- A pointer contains the address of a variable
  - allowing access to the variable indirectly



Copyright ©1994-2011 CRS Enterprises

58

A pointer is a scalar data item that contains the address of some other variable. In the example above a variable count has the value 25000 and is stored at address 0x190A. The pointer count\_ptr has the value 0x190A which is the address of count. The address of count\_ptr is unimportant.

Pointers provide an indirect method of accessing data in C++. Applications typically make extensive use of pointers and are an essential tool in the programmer's repertoire.

# Pointers

- **Pointers allow us to...**
  - allocate memory dynamically
  - change values passed as arguments to functions
    - call by reference
  - deal with arrays concisely and efficiently
  - effect whole-structure operations
  - represent complex data structures
    - such as linked lists, trees, stacks
- **Pointers are essential for writing efficient programs**
  - but common source of buggy code
  - uninitialised pointers can crash programs

It is essential to get to grips with the concept of pointers. It is the only way that some techniques can be achieved.

Some of the use of pointers include:

**allocation of arbitrary amounts of memory at run time.**  
**call by reference semantics when working with functions.**  
**deal with arrays concisely and efficiently**  
**effect whole-structure operations**  
**represent complex data structures such as linked lists, trees and stacks**

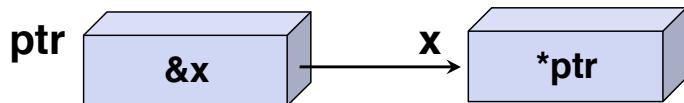
The downside of using pointers is that they tend to result in code that is difficult to read and maintain and are often the source of obscure and intermittent bugs.

## Declaring Pointers

```

int* int_ptr;      // pointer to int
long* lp;          // pointer to long
double* dp;        // pointer to double
int i,j,k,*ip;    // i, j and k are integers
                   // *ip is a pointer to int
long** q = &lp;   // pointer to pointer to long

```

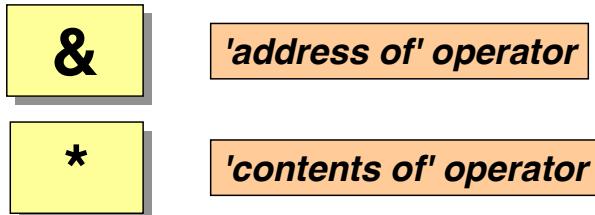


An individual pointer is constrained to contain addresses of only one type. The type is specified in the declaration. Therefore, a pointer is declared as a ‘pointer to type’. The position of the \* in the declaration is important, but any amount of white space can be used. It is recommended that the \* should be written next to the pointer’s type and that pointers and non-pointers should not be declared in the same statement. If you do mix pointer and non-pointers in the same declaration it will not be possible to place the \* next to the pointer’s type (see the fourth declaration above).

Note that the last example includes an initialization. The & operator is used to generate the address of its operand. The \* operator is used to dereference the pointer (retrieve the data at the end of the pointer).

## Pointer Operators

- Two unary operators used with pointers



The & operator generates the address of a previously-declared variable or const.

The \* operator accesses the value contained at the address contained in the pointer.

These two operators are inverse operators.

## Declaring Pointers

**int\* px;**

- **\*px can be thought of as an int**
  - may be used anywhere an integer variable can be used

```
int main()
{
    int x = 10;
    int y;
    int* px = &x; // initialize px to point to x

    x = *px + 1; // increment x
    y = *px / 2 + 10 - 7;
    if(*px > 10)
        printf("*px is %i\n", *px);
}
```

*A pointer is bound to a particular type*

Copyright ©1994-2011 CRS Enterprises

62

To declare a pointer to an int use

**int\* px**

The declaration restricts the pointer such that it can only ever point to an int. To dereference the pointer use **\*px**. Thus you can think of **\*px** being an alias for the variable at the end of the pointer. The alias **\*px** can be used wherever it is legal use an int.

## Using & and \*

- Fill in the values of a, b, c and p

```
int main()
{
    int a = 46;
    int b = 19;
    int c;
    int* p = &a;

    c = *p;
    p = &b;
    c = *p;
    b = 77;
    c = *p;
}
```

*Address*

a	<input type="text"/>	1246
b	<input type="text"/>	1250
c	<input type="text"/>	1262
p	<input type="text"/>	1272

The actual addresses used in this example are not important. Work through this example in a debugger to check out what happens to b, c and p.

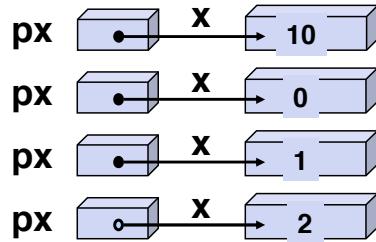
## Manipulating Pointers

- Pointer dereferences can occur on the left-hand-side of assignments

```
int main()
{
    int x = 10;
    int* px = &x;

    *px = 0;
    *px += 1;
    (*px)++;

    return 0;
}
```



Copyright ©1994-2011 CRS Enterprises

64

Since `*px` can be thought of as an integer variable, you can assign to it! In the above example `px` points at `x` and therefore `*px` is an alias for `x`. The three statements involving `*px` are equivalent to

```
x = 0;
x += 1;
x++;
```

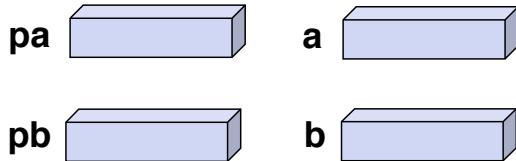
Note the parenthesis in the last example.

## Data and Pointers

- Distinguish between
  - $*p$  (an integer) and  $p$  (a pointer)

```
int main()
{
    int a = 7;
    int b = 3;
    int* pa = &a;
    int* pb = &b;

    *pa = *pb;
    pa = pb;
}
```


Copyright ©1994-2011 CRS Enterprises
65

Pointers are data items whose values are addresses. If a pointer is accessed for manipulation, it will be an address that is being processed. Once a pointer has been dereferenced using the `*` operator, it is the data pointed at by the pointer that is being processed. The compiler will warn you about mismatches such as `*pa = pb`.

The statement

`*pa = *pb`

is one of integer assignment, as both `*pa` and `*pb` are ints. The effect is to assign 3 to a.

However, the statement

`pa = pb`

is one of pointer to integer assignment, as both `pa` and `pb` are pointers to ints. The effect is to assign b's address to `pa`, i.e. both pointers now point to b.

## \* operator

- The \* operator has different meanings in different contexts

– often leads to confusing code

```
int main()
{
    int i = 3, j = 2, k;
    int* p = &i;
    int* q = &j;

    k = *p**q;
    k = (*p)++**q;
    k = *p/*q;
}
```

Copyright ©1994-2011 CRS Enterprises

66

Beware that the \* operator has different meanings in different contexts and this often leads to confusing code. Consider

$k = *p**q;$       this means k equals  $*p$  multiplied by  $*q$   
 $k = (*p)++**q;$  this means k equals  $*p$  multiplied by  $*q$  and  $*p$  is incremented  
 $k = *p/*q;$       this doesn't even compile - the intention is to assign  $*p$  divided  
                       by  $*q$  to k, but /\* is the start of comment sign

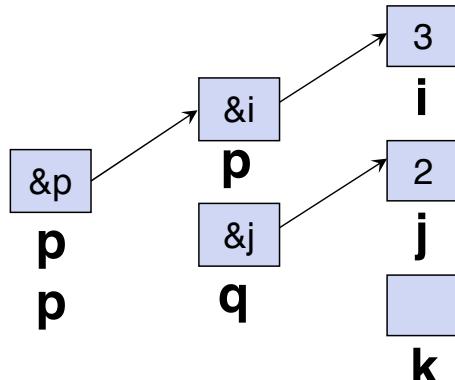
## Pointers to Pointers

- declare as `int**`

```
int main()
{
    int i = 3, j = 2, k;
    int* p = &i;
    int* q = &j;
    int** pp;

    pp = &p;
    printf("%i", **pp);
    p = q;
    printf("%i", **pp);

    k = *p * **pp;
}
```



Copyright ©1994-2011 CRS Enterprises

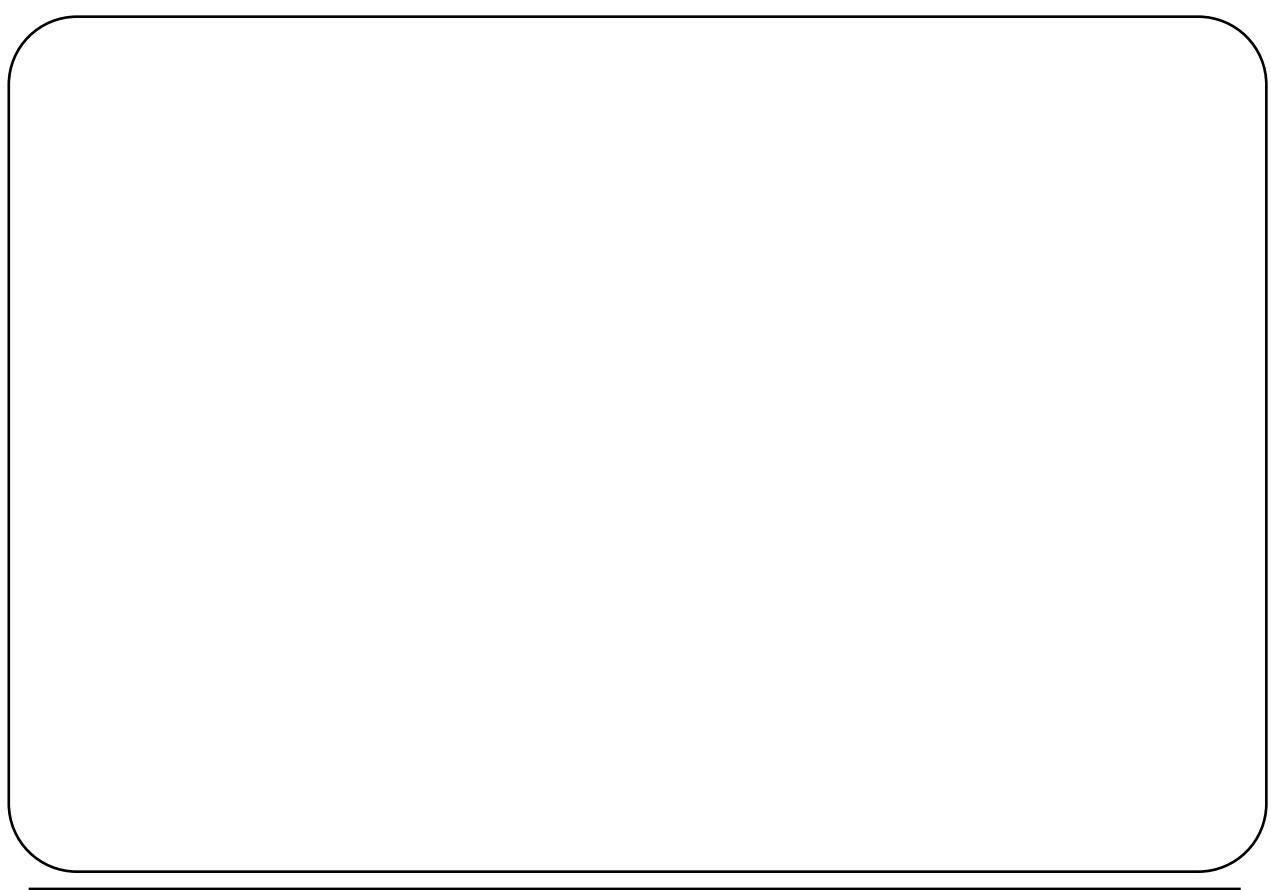
67

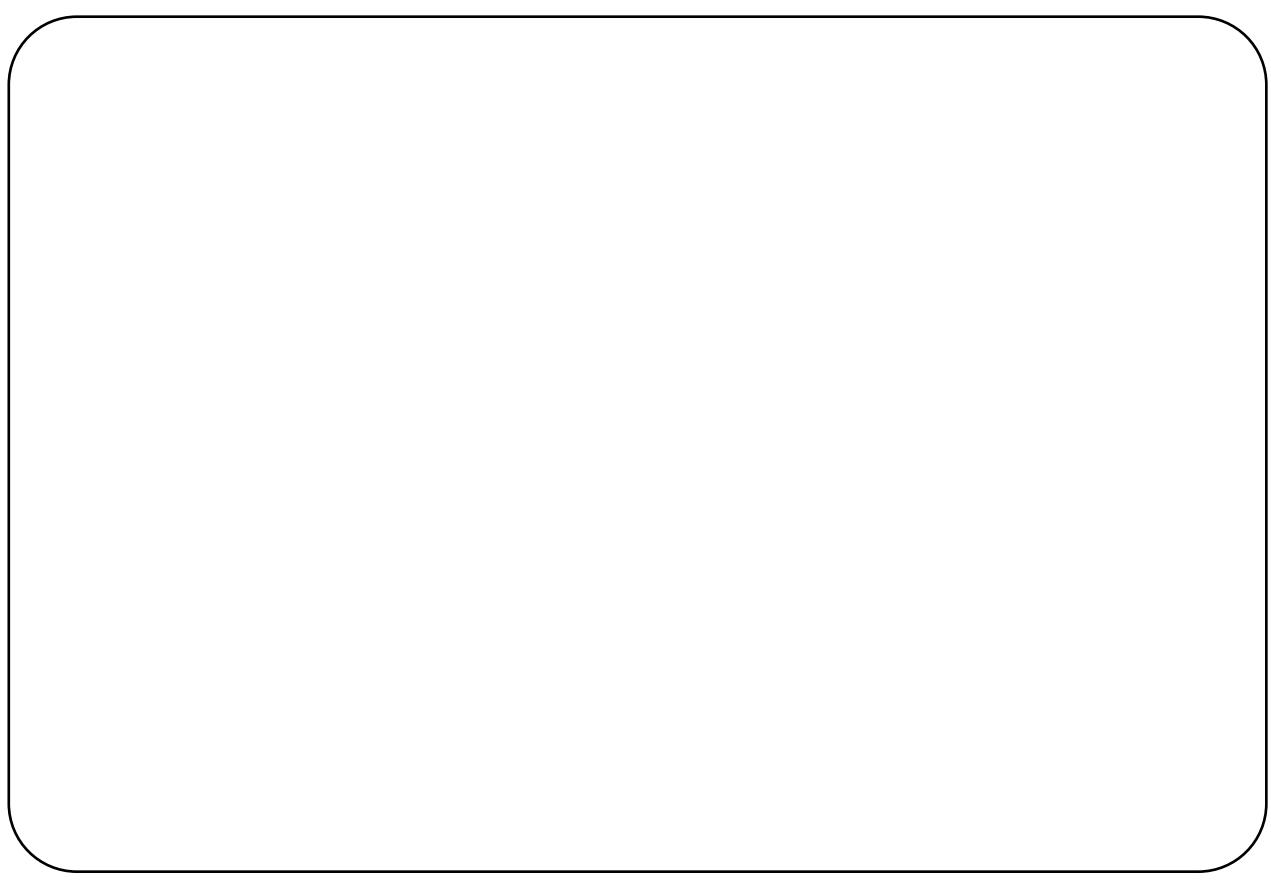
Pointers can be used to point to other pointers. The declaration `int** pp` should be read as `pp` is a pointer to an `int*` (itself a pointer to an `int`). As with other pointers the runtime expression `**pp` is an alias to the `int` at the end of the pointer chain.

Note the last expression

`k = *p * **pp`

`*p` is an alias for the data pointed at by `p` and `**pp` is an alias for the data pointed at by the pointer chain starting with `pp`. These two integers are multiplied together and stored in `k`. Try using a debugger to check what value gets assigned to `k`.





# 6

# Arrays

- **Arrays**
  - declaring
  - initialising
  - accessing
- **Functions**
  - passing arrays as parameters
  - library functions
- **Advanced Arrays**
  - arrays of characters
  - multi-dimensional arrays



# 6

Copyright ©1994-2011 CRS Enterprises

71

In C and C++, arrays are very closely related to pointers. In this chapter we investigate how to declare, initialize and access arrays and how to passing arrays as parameters. In the course of this investigation, we will see how pointers related to arrays. We will also see how arrays of characters are treated as a special by the compiler.

The chapter concludes with a short discussion of multi-dimensional arrays.

## Declaring Arrays

- General form of an array declaration

```
type name[size];
```

- The size specifies the number of elements in the array.  
Each element is of the specified type

```
int numbers[10];
int main(void)
{
    double vector[100];
    char   line[132];
    ...
}
```

The diagram shows three declarations of arrays within a code block. A yellow box highlights the declarations. Three arrows point from the end of each declaration to the right, each followed by a descriptive text. The first arrow points to 'numbers[10]', with the text 'declares numbers as an array of 10 ints'. The second arrow points to 'vector[100]', with the text 'declares vector as an array of 100 doubles'. The third arrow points to 'line[132]', with the text 'declares line as an array of 132 chars'.

- Array elements are stored in contiguous memory

Arrays are declared with a fixed size and are strongly typed (they can only contain a single type). Arrays can be declared as global or local variables.

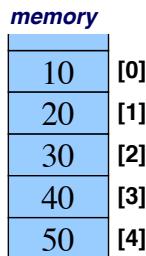
Note that it is not possible to change the size of an array at run time. However, it is possible to emulate dynamic arrays using pointers (see a later chapter).

## Declaring and Initialising Arrays

- **Arrays may be initialised when declared**

```
int numbers[5] = { 10, 20, 30, 40, 50 };

int main(void)
{
    double vector[100] = { 3.8, 12.6 };
    char name[] = { 'G', 'e', 'o', 'r', 'g', 'e', '\0' };
    ...
}
```



Copyright ©1994-2011 CRS Enterprises

73

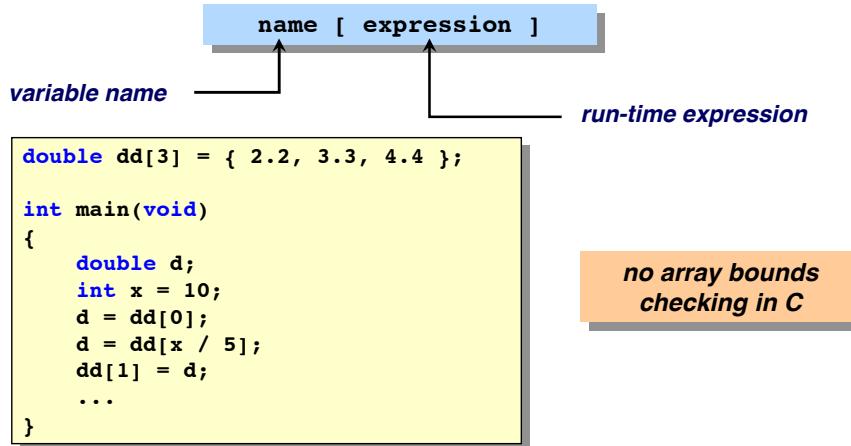
Just like scalar data, an array can be initialised at declaration time. The syntax is similar to that of scalar data with the = operator is used. Normally you will give an initial value to each element, but partial initialisations are allowed. The initialisers must be of the correct type, separated by commas and enclosed in {}.

If there are too few initial values, sometimes referred to as a ‘short-fall’, the remaining values are set to zero. Too many give rise to a compiler error.

On occasions, the size in the declaration can be left for the compiler to fill in. This is achieved by leaving the size blank and by supplying the correct number of initial values. Note that some compilers do not guarantee to set uninitialised elements to zero even though it is part of the C89 standard.

## Accessing Array Elements

- Elements can be accessed using run-time expressions



Copyright ©1994-2011 CRS Enterprises

74

Array indexing always starts at 0. Thus an array of N elements is indexed using the range 0 to N-1. The index expression must be integral.

Unlike languages like Java and C#, the C compiler does not ensure accesses beyond the end of an array are prohibited; we say that there is no array bounds checking in C. This can be the source of very difficult to diagnose bugs and is a major drawback with the design of the language.

Since the compiler does help, it is vital to build bound checking into your code. The onus is on you!

## Exercise

- C/C++ does not support *complete array operations*
- Arrays are manipulated element by element

```

int a[10] = { 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };

int main()
{
    int b[10];
    int i;
    for (i = 0; i < 10; i++)
    {
        // b = 0;
        // b = a;
        // print b
        // input b
    }
}

```

How do we  
implement  
these  
operations?

Copyright ©1994-2011 CRS Enterprises

75

Unlike languages like Fortran, PERL or Python, C/C++ does not provide any support for complete array operations such as assignment. Instead, everything has to be achieved through individual element access.

The examples above are implemented using loops:

- 1) **b = 0**  
`for (i = 0; i < 10; i++) b[i] = 0;`
- 2) **b = a**  
`for (i = 0; i < 10; i++) b[i] = a[i];`
- 3) **print a**  
`for (i = 0; i < 10; i++) printf("%i", a[i]);`
- 4) **input b**  
`for (i = 0; i < 10; i++) scanf("%i", &b[i]);`

## Arrays and Functions

- To pass an array to function

- use the name of the array
- arrays passed by reference

```
void change(double [ ]);
int main(void)
{
    double vector[5] =
        { 0.0, 1.1, 2.2, 3.3, 4.4 };
    change(vector);
    printf("%lf %lf", vector[0], vector[3]);
    return 0;
}
void change(double da[])
{
    da[0] = 3.14159;
    da[3] = da[2] + da[4];
}
```

memory
0.0
1.1
2.2
3.3
4.4

Copyright ©1994-2011 CRS Enterprises

76

When arrays are passed as parameters to functions only the *address of the first element in the array* is passed. This address is a pointer to the array. The function receives a copy of the pointer in line with C's call by value semantics. This has several implications.

The address of the first element of the array tells the function nothing about the size of the array being passed. In fact the function has no way of determining the size of the array passed (unless the size is passed as a separate parameter). If you use the `sizeof` operator it always returns 4 bytes - the size of the pointer being passed!

Since the function can't tell the size of the array, you omit the size in the prototype:

`void change(double []);`

Alternatively you can write

`void change(double*);`

Since access to the array being passed is through a pointer, the changes to the array inside the function will modify the original array. This is often described as "*arrays are passed by reference*", but in reality the pointer is passed by value.

Finally since the function has no way of knowing the size of the array being passed, you must be very careful not to write beyond the bounds of the array - best practice is to pass a second parameter defining the size the array.

## Example

```

void initialise_array(int [ ], int);

int main(void)
{
    int x[100];
    int y[7];
    initialise_array(x, sizeof(x));
    initialise_array(y, 7);
    return 0;
}

/* Initialise array to zeros */
void initialise_array(int ai[ ], int size)
{
    int i;
    for (i = 0; i < size; i++)
        ai[i] = 0;
}

```

Copyright ©1994-2011 CRS Enterprises

77

The examples above follow the recommended practice of passing the size of the array as a second parameter.

Note that `sizeof(x)` does return the number of bytes in the array (probably  $4 * 100$ ) because `x` really is an array. If we tried `sizeof(ai)` inside the function it would return 4 bytes, because despite appearances, `ai` is a pointer and not an array.

## Exercise

- How do we write `sum()`?

```

int sum(int[ ], int);
int main(void)
{
    int theArray[5] = { 1, 4, 9, 16, 25 };
    int total = 0;
    total = sum(theArray, 5);
    printf("Sum = %i\n", total);
    return 0;
}
int sum(int a[ ], int size)
{
}

```

*How do we  
implement  
this function?*

Copyright ©1994-2011 CRS Enterprises

78

The sum function can be implemented as follows:

```

int sum(int a[ ], int size)
{
    int i;
    int sum = 0;
    for(i = 0; i < size; i++)
    {
        sum += a[i];
    }
}

```

## Strings - Arrays of *char*

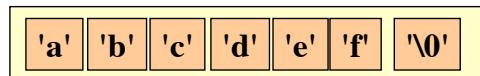
- No language support for strings (no ‘string’ type)

`string instrument;`

- Strings are implemented in C using arrays of *char*

`char instrument[64];`

- The convention is that strings must be terminated by the '\0' character



Strings are implemented in C as arrays of characters. In most languages, strings are variable size, but in C arrays are always fixed in size. In order to introduce some flexibility, the character array defines the maximum size of the string. Then when the string is given a value, a null character '\0' is placed at the end of the string, effectively defining an end marker inside the character array.

For example

`char instrument[64];`

defines a string with a maximum of 63 characters (1 byte is reserved for the null). If we copy the characters *a* through *f* into the array, only the first 6 characters of the array are used, the null appears in the 7<sup>th</sup> position and the remaining elements are unused.

## Arrays of Character

```
int main(void)
{
    char str[10] = {'H', 'e', 'l', 'l', 'o', '\0'};
    printf("%s world \n", str);
    return 0;
}
```

*%s in printf*

memory
72
101
108
108
111
0
?
?
?

A string can be initialised just like any other array, but remember to put the '\0' in at the end. The length of a string is the number of characters up to, but not including, the null character.

The *printf* function supports strings; %s can be used to output a string. The *scanf* function provides the corresponding support for string input.

## Language Support for Strings

- The language supports string literals enclosed in double quotes

```
int main(void)
{
    char str[ ] = "world";
    printf("%s %s", "Hello", str);
    return 0;
}
```

A callout box with a grey border and a black arrow points from the text "string literals" to the two string literals "world" and "Hello" in the code snippet.

- String literals are effectively read-only character arrays

```
void modify(char bad[ ])
{
    bad[0] = 'J';
}
modify("Hello");
```

A black bomb icon with a red fuse is positioned to the right of the code snippet.

Constant strings (or string literals) can be created using double quotation marks. This technique has already been used for the format strings in both the *printf* and *scanf* functions. A string constant can be used to initialize a string. This is effectively a character-by-character copy, from the constant into the array. This is only valid for initialization, at runtime the string must be copied character by character.

Note that a string constant can also be output from *printf* using *%s*.

## C Runtime Library Functions

- Null terminated Text Strings

- `strlen`
- `strcpy`
- `strcmp`

- Binary Strings

- `memcpy`
- `memmove`
- `memcmp`

```
#include <string.h>

int main(void)
{
    int length;
    char s1[ ] = "Hi There";
    char s2[50];

    length = strlen(s1);
    printf("s1 is %d long\n", length);

    strcpy(s2, s1); ← Just like
                     s2 = s1;

    printf("%s world\n", s2);
    return 0;
}
```

Copyright ©1994-2011 CRS Enterprises

82

The C runtime library has extensive support for strings. The library provides standard functions to manipulate strings, such as `strlen`, `strcpy` and `strcmp`.

<b>strlen</b>	returns the length of a string
<b>strcpy</b>	copies strings
<b>strcmp</b>	compares strings

The library not only has support for null terminated strings, but also binary strings. Binary strings use the full extent of their character array and hence do not contain null characters. Binary strings can be used to represent untyped data in memory. Some of the more common binary string routines are:

<b>memcpy</b>	copy memory (non overlapping regions)
<b>memmove</b>	copies memory (possibly overlapping regions)
<b>memcmp</b>	compares memory

## Multidimensional Arrays

- C allows arrays of any number of dimensions to be defined:

```
type array_name[dim1][dim2]...[dimN];
```

e.g. `int table[8][6];`

- C stores arrays in right hand dimension order
  - column first for 2D arrays
  - opposite to how Fortran handles arrays

[0][0]
[0][1]
[0][2]
[0][3]
[1][0]
[1][1]
[1][2]
[1][3]

C allows arrays of any number of dimensions to be defined. A separate set of [ ] are used for each dimension.

If you are used to programming in Fortran, note that in C multi-dimensional arrays are stored in memory with the right most dimension changing fastest. In Fortran, the left most dimension changes the fastest.

## Initialising Multidimensional Arrays

- Multidimensional arrays can also be initialised

```

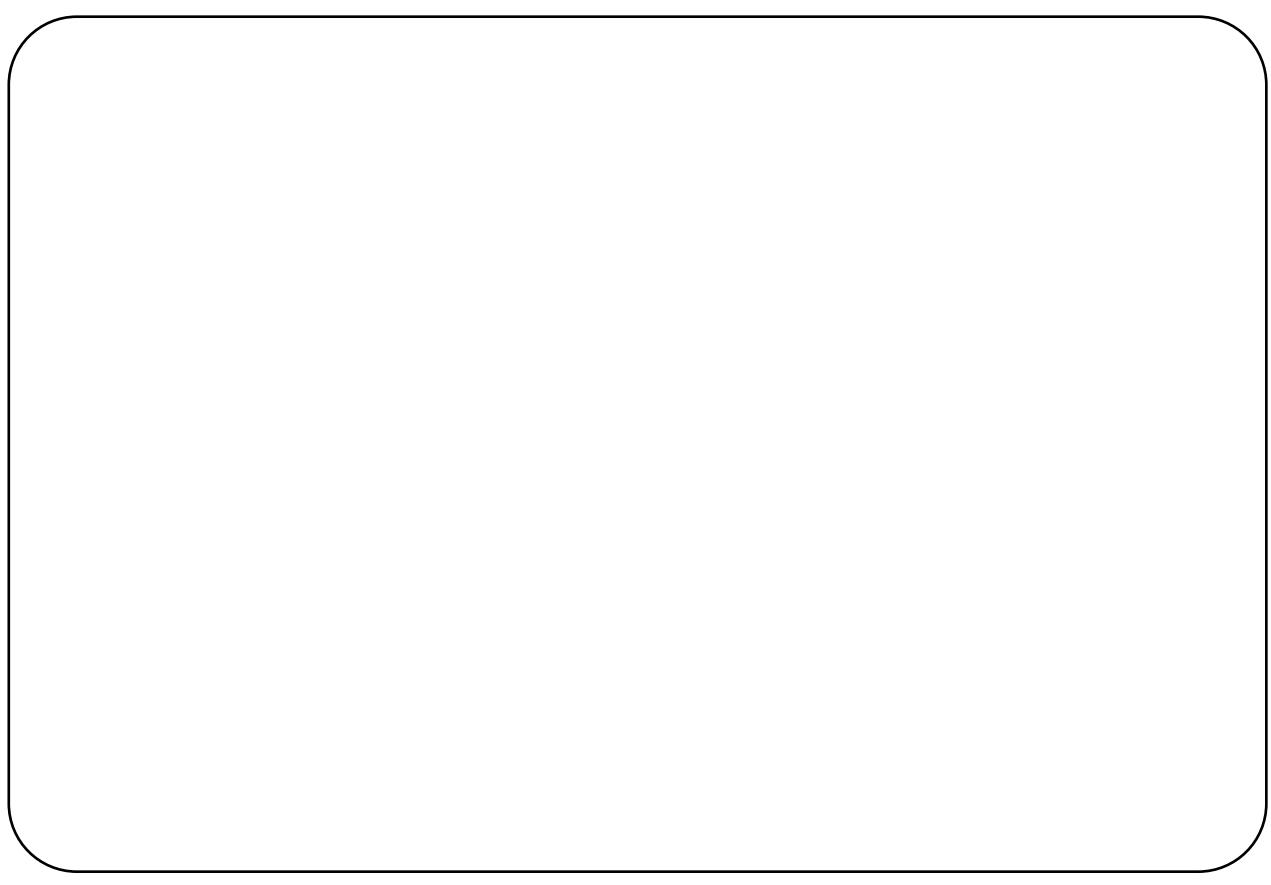
int results[100][5] =
{
    { 75, 95, 90, 84, 80 }, /* row 0 */
    { 100, 99, 100, 98, 99 }, /* row 1 */
    ...
    { 80, 67, 85, 79, 75 } /* row 99 */
};

double coordinates[ ][2] =
{
    { -2.5, 2.71 },           /* (x,y) point 0 */
    { 0.0, 0.0 },             /* (x,y) point 1 */
    { 4.0, 12.03 }            /* (x,y) point 2 */
};

```

Initialization of multidimensional arrays is performed using a set of {} for each dimension suitably nested. It is recommended that you layout initialization code as above for reasons of readability.

Note also that the compiler can calculate the row count for the second array from the initialization list.



# 7

# The Preprocessor

- **Basic Features**
  - Constants
  - Macro substitution
  - File inclusion
  - Conditional compilation
  
- **Advanced Features**
  - string sizing operator
  - #pragma

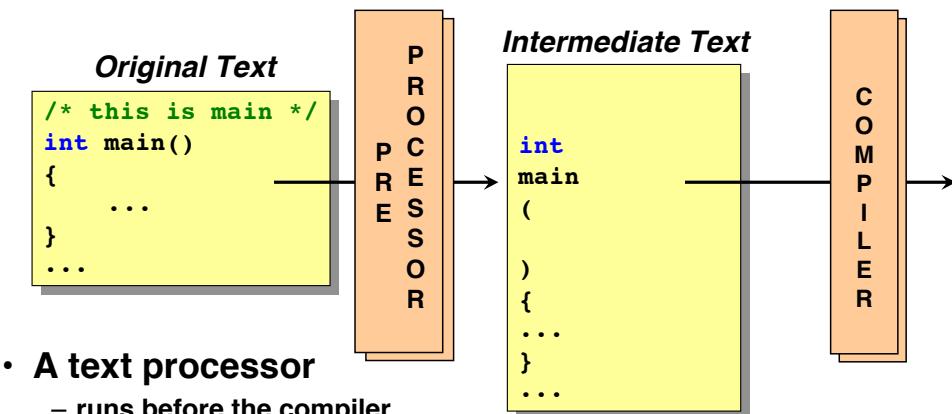


7

The C pre-processor (cpp) is the pre-processor for the C and C++ programming languages. In many C implementations, it is a separate program invoked by the compiler as the first part of translation. The pre-processor handles directives for source file inclusion (#include), macro definitions (#define), and conditional inclusion (#if). The language of pre-processor directives is agnostic to the grammar of C, so the C pre-processor can also be used independently to process other types of files.

The transformations it makes on its input form the first four of C's so-called Phases of Translation.

## What is a Preprocessor?



- **A text processor**
  - runs before the compiler
  - simple textual replacement
  - a tokeniser
  - strips out comments, ...

The preprocessor is separate from the compiler and performs limited text processing. You can think of the pre-processor as the first parse of the source code. Its capabilities are discussed in this chapter.

The main purpose of the pre-processor is to translate text according to a set of rules. The output from the preprocessor, becomes the input to the compiler. Normally the pre-processor output is held temporarily in memory and then discarded at the end of the compilation phase. However, using various compiler options, you can save the output as a temporary file with a .i extension or display the output on STDOUT.

## Preprocessor Directives

- Preprocessor directives may be embedded in C code
  - All directives start with #
- Symbolic Constants  
`#define`
- Include files  
`#include`
- Conditional Compilation  
`#ifdef, #ifndef, #else, #elif, #endif`
- String Concatenation  
`##`
- Quoting  
`#`

```
#include ...
#define ...
#ifndef ...

int main()
{
    ...
}
```

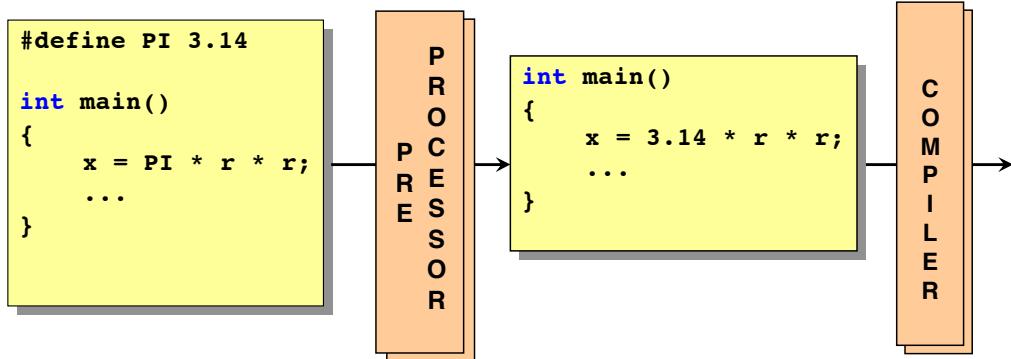
Pre-processor commands can be placed anywhere in the source file. Without exception, they begin with the # character. Syntax is similar, but not identical, to C. The pre-processor is regarded as part of the C environment and is controlled by the ISO standard. The following directives are supported:

```
#define and #undef
#include
#if #elif #else #endif
#ifndef #elif #else #endif
#endif
#line
#pragma
#error
```

As well as these, there are other facilities: trigraphs, predefined identifiers and the two operators, # and ##.

## #define

- A **#define** control line can be used to give a symbolic name to some text



- By convention, symbolic constants are in upper case

One of the main uses of the pre-processor is to give symbolic name to constants. By convention, preprocessor symbols are written as upper case. In the example above, the symbol PI is replace by the text 3.14. Note that the pre-processor only works with text; it is unaware of the concept of numbers, arrays and most other C artifacts.

The **#undef** directive is used to wipe out a previously **#defined** name. For example:

**#undef PI**

This is usually used when the symbol is to be given a new value in the source that follows. The symbol can be any legal C identifier.

Note: preprocessor directives are terminated by the end of line unless the \ continuation character is used.

## More #defines

```
#define BUFFER 100
int main()
{
    double buffer[BUFFER];
    int i;
    ...
    for (i = 0; i < BUFFER; i++)
        buffer[i] = 0.0;
    ...
}

#define VAT 17.5
#define TAX 1.5 + VAT
int main()
{
    double amount;
    ...
    → amount = TAX * 21.50;
    ...
}
```

what does this expand to

Copyright ©1994-2011 CRS Enterprises

91

By using the pre-processor to define symbolic constants, it becomes much easier to maintain code. For example suppose a buffer size is used repeatedly throughout a program. If the buffer size needs to be changed, then only the one #define needs to be modified.

Because the pre-processor effectively performs a single parse, the scope of its #define tokens are global and take effect on the line after the definition. This means that you may define symbols using symbols already defined. However, you must be careful with precedence. Brackets are nearly always required. For example, with the definition:

**#define TAX 1.5 + VAT**

Without the brackets the preprocessor will expand

**amount = TAX \* 21.50;**

into

**amount = 1.5 + VAT \* 21.50;**

and C precedence rules mean that 1.5 is not included in the multiplication. But with brackets

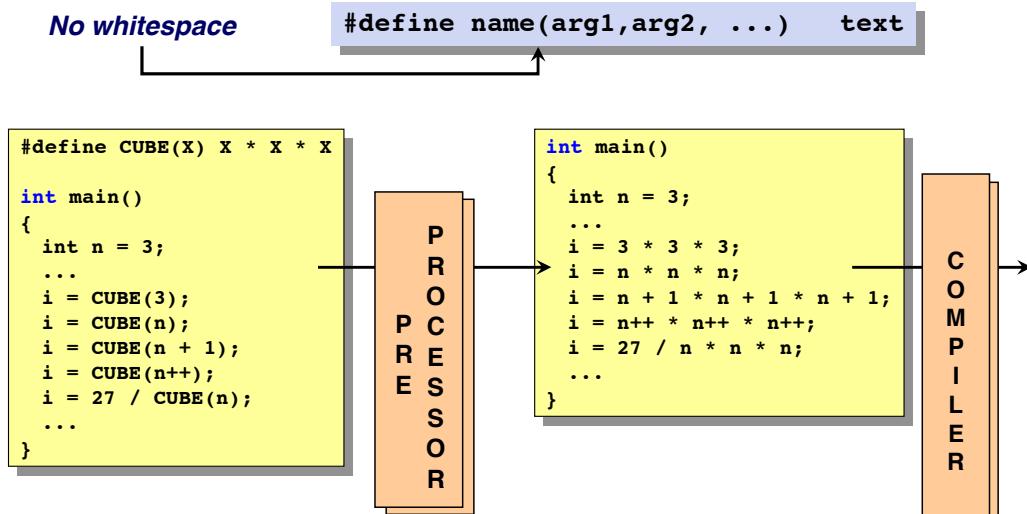
**#define TAX (1.5 + VAT)**

the expansion becomes

**amount = (1.5 + VAT) \* 21.50;**

## Function Macro

- #define can take parameters



Copyright ©1994-2011 CRS Enterprises

92

#defines can take parameters and behave like functions in C. Such functions are called macros. In the example above we should have defined `CUBE(X)` as

`((X) * (X) * (X))`

with brackets everywhere. Observe the expansions that occur without the brackets. The last three examples are all incorrect because of precedence rules. Remember, if you use a #define put in brackets everywhere!

## Function or Macro

- Preprocessor runs before the compiler
  - functions with the same name as a macro will be hidden
- Preprocessor Macros
  - are typeless
- Functions
  - are typesafe

**bracket fools  
the preprocessor**

```
#define CUBE(X) X * X * X
int CUBE(int x);

int main()
{
    ...
    result = CUBE(3);
    result = CUBE(53.92);
    result = (CUBE)(3);
    ...
}
```

You can define functions as macros or as true C functions. Macro definitions are typeless, unlike their C counterparts and this means the macro can be called with ints or doubles (the pre-processor treats both as text).

Note that if you define a C function and a macro with the same name, calls such as

**result = CUBE(3)**

appear ambiguous, but the macros gets called (effectively masking the C function) because the pre-processor runs as the first pass of the compiler. If you really want to call a C function with the same name as a macro, place brackets round the function name to fool the pre-processor (they are ignored by the compiler):

**result = (CUBE)(3)**

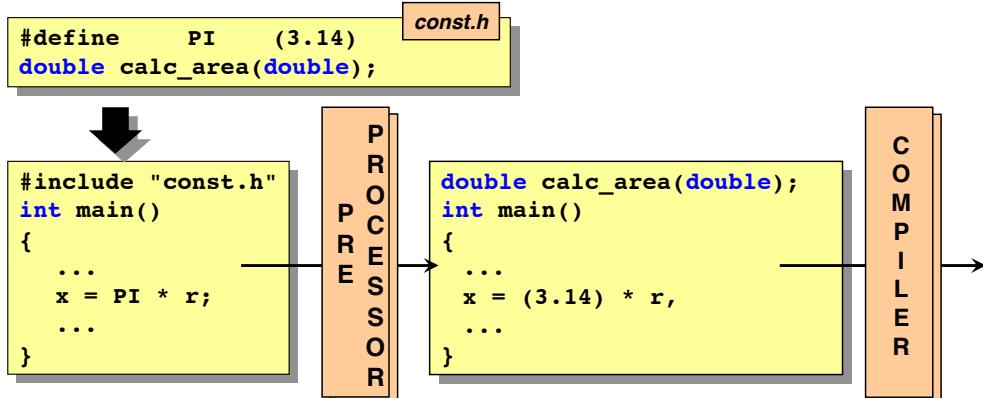
## #include

- Includes the entire contents of the file specified

**<>** for system header files

**""** for programmers header files

```
#include <filename>
#include "filename"
```



Copyright ©1994-2011 CRS Enterprises

94

The other main uses of the pre-processor is `#include`. A `#include` effectively pastes the contents of an entire file into your source code. You can use `#include` in one of two ways. If the filename is enclosed in angle brackets, the search for header file looks through standard directories (as specified by the compiler). If the filename is enclosed in double quotes, the compiler uses the name as an absolute or relative pathname starting from the current directory. If that search fails, the compiler adopts the search rules for the angle-bracket notation.

## Header Files

- Include files have a .h extension

– C++ drops the .h when using libraries and uses <cstring> instead of <string.h> etc

```
#ifndef MYHEADER_H
#define MYHEADER_H
...
#define     CHAR_BIT    (8)
#define     INT_MAX    (+32767)
#define     INT_MIN    (-32768)
#define     SHRT_MAX   (+32767)
#define     SHRT_MIN   (-32768)
...
void myFunction(int, int);
...
#endif
```

A header file is an ordinary text file. It contain text to be #included in your source file. You are advised to adhere to certain rules:

- 1) do not include function definitions. This may lead to illegal duplicate function definitions and the linker will fail.
- 2) do not include global data definitions. This will lead to separate global definitions in each compilation module, i.e. multiple definitions.

Header files can be used to contain #defines and function prototypes. You can also nest #includes.

As a precaution against multiple inclusion with nested #includes, you are advised to wrap all header files with

```
#ifndef unique-identifier
#define unique-identifier
    content-of-header-file
#endif
```

This will ensure the definitions are only defined once per compilation module, even if the header file is include twice. The first time the header file is called, the unique-identifier will be undefined and therefore the next statement will define it and all the lines up to the #endif will be processed. The next time the header file is encountered the unique-identifier will be defined and therefore all lines up to #endif will be skipped.

## #if

- This directive allows lines of source text to be conditionally included or excluded from compilation
- General form is...

```
#if expr1
    group_of_lines_1
#elif expr2
    group_of_lines_2
#else
    group_of_lines_3
#endif
```

*Only one of these groups is sent to the compiler*

- Useful for isolating code dependent on client, OS, version, etc.

The #if construct can be used for conditional compilation. Note that the pre-processor uses syntax that is slightly different from that used by the C language itself.

## Advanced Facilities

- Compiler concatenates adjacent string literals...

"Hello" " World" → "Hello World"

- Text preceded by # is made into a string literal...

#define MAKE\_STRING(s) #s  
 MAKE\_STRING(x > 0) → "x > 0"

- Several identifiers are predefined ...

\_\_LINE\_\_ current source line number  
\_\_FILE\_\_ string literal - name of file

Another feature of the pre-processor is to splice together adjacent string constants. "Hello" "World" will be joined to become a single literal "Hello World" (note that the space is a character from the second string). This mechanism is useful when writing long string constants, usually in printf statements.

Creating string literals with the pre-processor requires the use of another feature; the # operator. The # operator creates a string from its single argument.

Several predefined symbols can be used. To avoid conflicts with your symbols, the preprocessor symbols all begin and end with a double underscore.

## Stringizing Operator

- The # operator can be used to convert macro parameters into strings
  - encloses parameters in double quotes

```
#define VALUE(expr) printf("%s = %i\n", #expr, (expr))  
...  
VALUE (i * j + 5);
```

```
printf("%s = %i\n", "i * j + 5", (i * j + 5));
```

If the # stringizing operator is used before a macro parameter, that parameter is turned into a string (it is surrounded by double quotes). This is handy when writing debug macros such as the example shown above.

## #pragma

- **#pragma allows compiler specific definitions**
  - no standard that specifies what pragmas should be supported by C compilers
- **Unrecognised pragmas are ignored**

```
#pragma check_pointer(off)

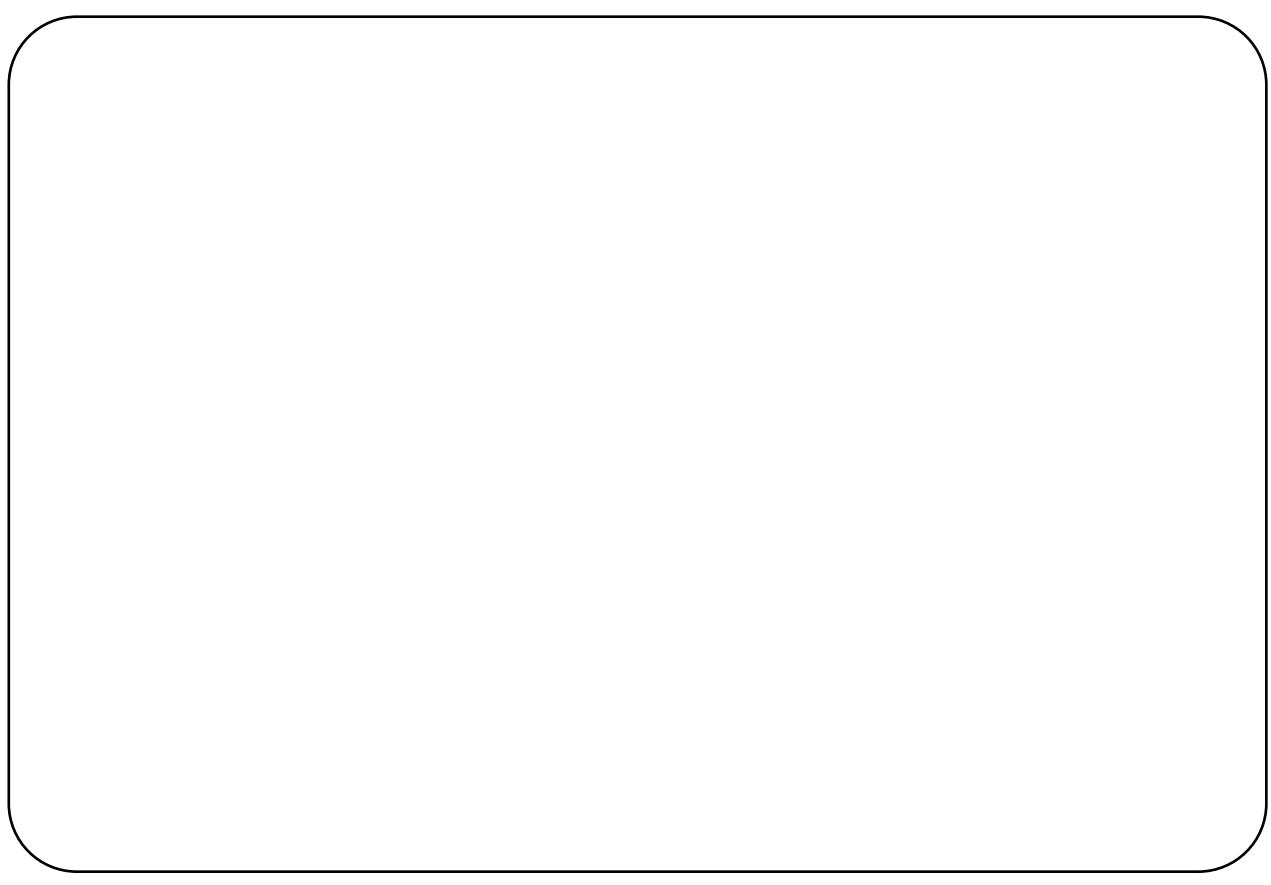
for (p = a, end = &a[SIZE]; p < end; p++)
    *p = 0;

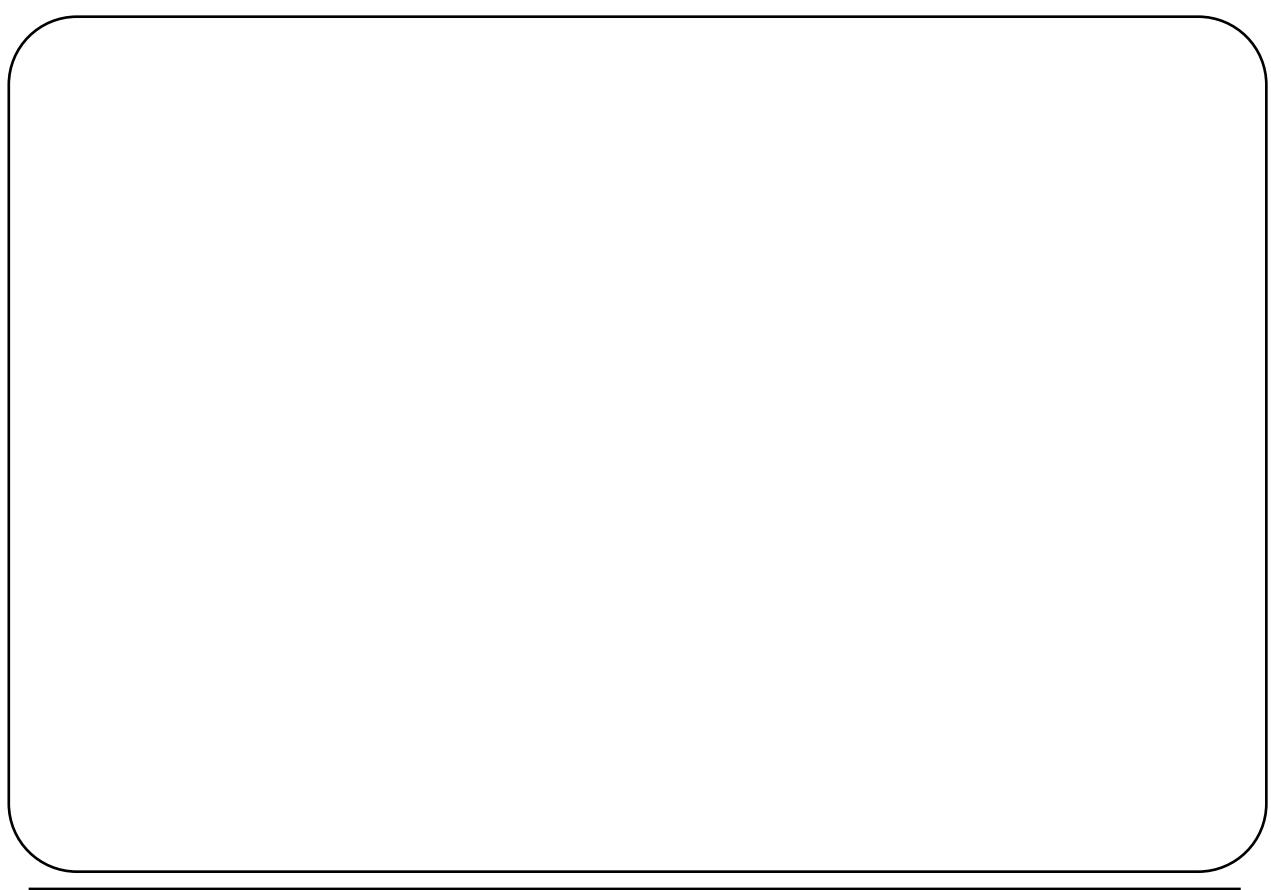
#pragma check_pointer(on)
```

The last feature of the pre-processor we will discuss is #pragma. This directive is read by the pre-processor, but affect either the compiler or the optimiser and the way it produces code. Pragmas provide a way of guaranteeing that code is compiled with certain options in force, rather than leaving things up to the user (who may not know what flags to provide to the compiler). There is no standard list of pragmas, each compiler defines its own set of pragmas.

As an example, the following list shows some of the more commonly-used pragmas supported by the Microsoft compiler:

<b>#pragma check_pointer</b>	enable/disable pointer checking
<b>#pragma check_stack</b>	enable/disable stack checking
<b>#pragma function</b>	specifies that calls to a function will take place as a conventional function call
<b>#pragma intrinsic</b>	specifies that calls to a function will be expanded inline
<b>#pragma pack</b>	controls structure packing





# 8

## Input and Output

- **Output using cout**
- **Input using cin**
- **istream and ostream classes**
- **Formatting output**
- **Handling input errors**



8

Input and output in C++ have been completely changed from that used in C. The C run time library routines such as *printf* and *scanf* have been replaced with **typesafe** input/output.

This chapter shows how to use typesafe I/O via *ostream* and *istream* classes. Standard *cin* and *cout* objects are introduced and we discuss how these objects are used to perform formatted input and output. The new typesafe I/O represents a considerable improvement on C's input/output facilities.

## Output using cout

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

int main ()
{
    int minutes = 55;
    double distance = 231.75;
    char message[ ] = "Hello";

    cout << "Minutes = " << minutes << endl;
    cout << "Distance = " << distance << endl;
    cout << "Message is: " << message << endl;
}
```

Input and output in C++ have been completely changed from that used in C. The C run time library routines such as printf and scanf have been replaced with typesafe input/output.

This chapter shows how to use typesafe I/O via ostream and istream classes. Standard cin and cout objects are introduced and we discuss how these objects are used to perform formatted input and output. The new typesafe I/O represents a considerable improvement on C's input/output facilities.

The program above prints numbers and text on the terminal screen. The header file iostream contains all the definitions for the C++ input/output classes. cout is an object belonging to the output class ostream.

The << operator is normally the shift left operator. However, C++ allows us to redefine operators on a class by class basis. The ostream class has redefined the meaning of this operator from shift left to print (see the chapter on operator overloading).

The symbol endl is a manipulator that prints a newline and flushes the cout object to the screen. The ostream class defines several manipulators to facilitate formatting output.

## Input using cin

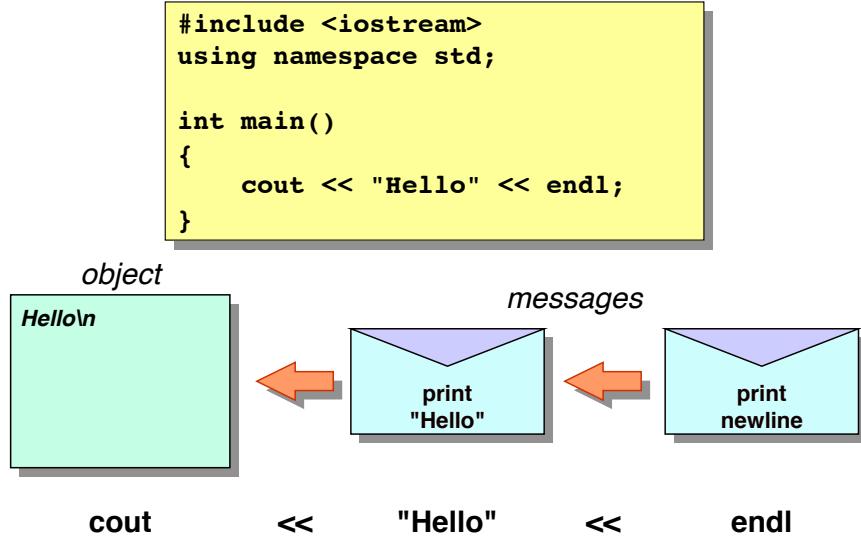
```
#include <iostream>
using namespace std;

int main()
{
    int number;
    double distance;
    char message[20];

    cout << "Please enter data: ";
    cin >> number;
    cin >> distance;
    cin >> message;
}
```

The `cin` object is used to accept input. This example shows how to input integers, doubles and character strings. Note that we no longer need to use the format codes of `printf`; the input is typesafe.

## Output Objects



Copyright ©1994-2011 CRS Enterprises

106

The `cout` object should be considered as a buffer. The `ostream` class to which it belongs defines all the messages you can send to this object; some messages are disguised as the `<<` operator (this will be explained in the section on operator overloading). The code fragment

**`cout << "Hello"`**

sends the message add the string "Hello" to your buffer to the `cout` object. Similarly:

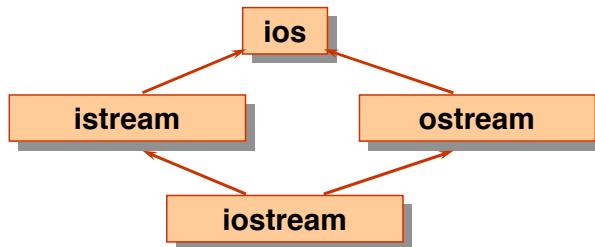
**`cout << endl`**

sends the special manipulator `endl` to the buffer. The manipulator inserts a newline character into the buffer and then flushes the buffer (forces the buffer contents to be sent to the screen).

Note that `cout` can accept multiple cascaded messages:

**`cout << "Hello" << endl;`**

## Input/Output Streams

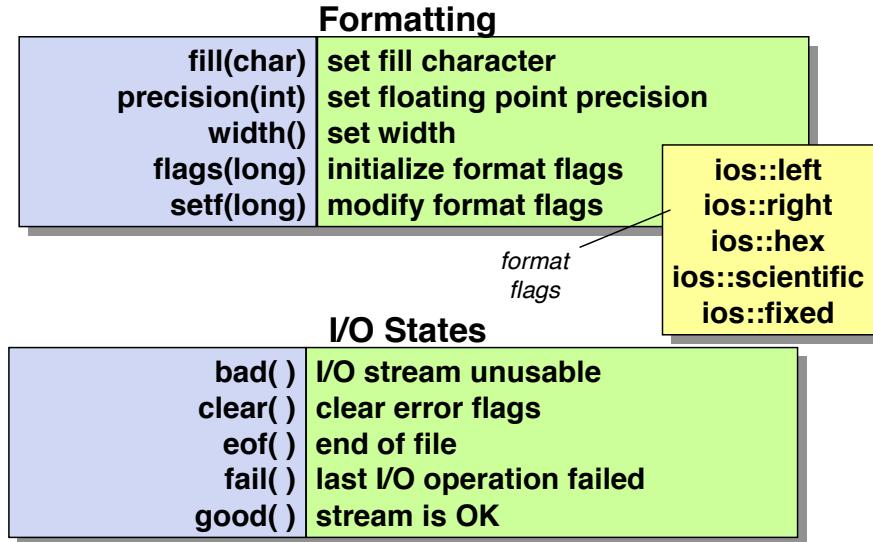


Object	Class	Equivalent C Stream
cin cout cerr clog	istream ostream ostream ostream	stdin stdout stderr (unbuffered) stderr (buffered)

The C++ input/output library is implemented as an inheritance hierarchy. The base class is ios which contains all the formatting operations that are common to input and output. The istream class defines all the input methods and the ostream class defines the output methods. The iostream class can be used for input and output.

The 4 objects: cin, cout, cerr and clog are created as part of the start up code for main. These objects are analogous to the stdin, stdout and stderr buffers of C. Note that cerr is used for immediate error reporting, whereas clog is used for logging errors to disk files.

# Formatting Input/Output



The ios, istream and ostream classes define formatting methods, useful enumerated constants and methods that determine the state of a stream.

The formatting methods allow the input and output to be precisely formatted. Some methods will require use of enumerator constants defined in the ios class. The state methods are used to check for errors. For example, if an error has occurred on the previous input then cin will be set to the fail state. cin will not be able to process new input until is has been sent the clear message. This allows the programmer to correct the problem before continuing. If a stream is in the bad state, it is corrupted and cannot be used again.

## Output Width

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

int array[4] = { 1, 12, 123, 1234};

int main ()
{
    for (int i = 0; i < 4; i++)
    {
        cout.width(6);
        cout << array[i] << endl;
    }
}
```



Copyright ©1994-2011 CRS Enterprises

109

The `width` method sets up the output width (in characters) for the next output message. The width setting only remains in force for one output operation after which the default width setting applies.

The message:

**`cout.width(6)`**

sets the width for both numbers and strings. The output is padded to the left by the default padding character (a space). The padding character can be changed by sending `cout` the message:

**`cout.fill('#')`**

The message:

**`cout.width(0)`**

is used to request a default width (output as many characters as required, but do not pad out the output).

## Output Precision

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

int main()
{
    double number = 5.6789;
    cout.setf(ios::fixed);

    for (int i = 1; i <= 4; i++)
    {
        cout.width(8);
        cout.precision(i);
        cout << number << endl;
    }
}
```



Copyright ©1994-2011 CRS Enterprises

110

The precision method sets up the number of decimal places used for the next output message (the final decimal place is rounded to the closest digit). The precision setting only remains in force for one output operation.

The message:

**cout.precision(0)**

is used to request a default precision (6 decimal places).

The iostream library will output a number in fixed point format unless the number is large.  
The messages:

**cout.setf(ios::fixed)**

**cout.setf(ios::scientific)**

can be used to force fixed point or scientific notation.

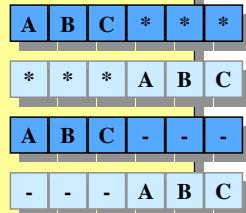
## Output Justification

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

void Print(char fill, long format);

int main ()
{
    Print('*', ios::left);
    Print('*', ios::right);
    Print('-', ios::left);
    Print('-', ios::right);
}

void Print(char fill, long format)
{
    cout.fill (fill);
    cout.width (6);
    cout.setf(format, ios::adjustfield);
    cout << "ABC" << endl;
}
```



Copyright ©1994-2011 CRS Enterprises

111

The setf method is used to set various flags. This method is often used to select left or right justification:

<b>cout.setf(ios::right, ios::adjustfield)</b>	<b>right justification</b>
<b>cout.setf(ios::left, ios::adjustfield)</b>	<b>left justification</b>

Note that the second parameter (ios::adjustfield) must be present for the first parameter to take effect. This is an unnecessary complication in the design of setf.

The fill method is used to set the fill character (the default fill character is a space).

The above settings remain in force until explicitly changed.

## Coping with Input Errors

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

int main() {
    double number;
    cout << "Enter data" << endl;
    while(true){
        cin >> number;
        if (cin.eof()) break;
        if (cin.fail()) {
            cout << "Bad input" << endl;
            cin.clear();
            cin.ignore(10, ' ');
        } else {
            cout << "Good input: ";
            cout << number << endl;
        }
    }
}
```

Copyright ©1994-2011 CRS Enterprises

112

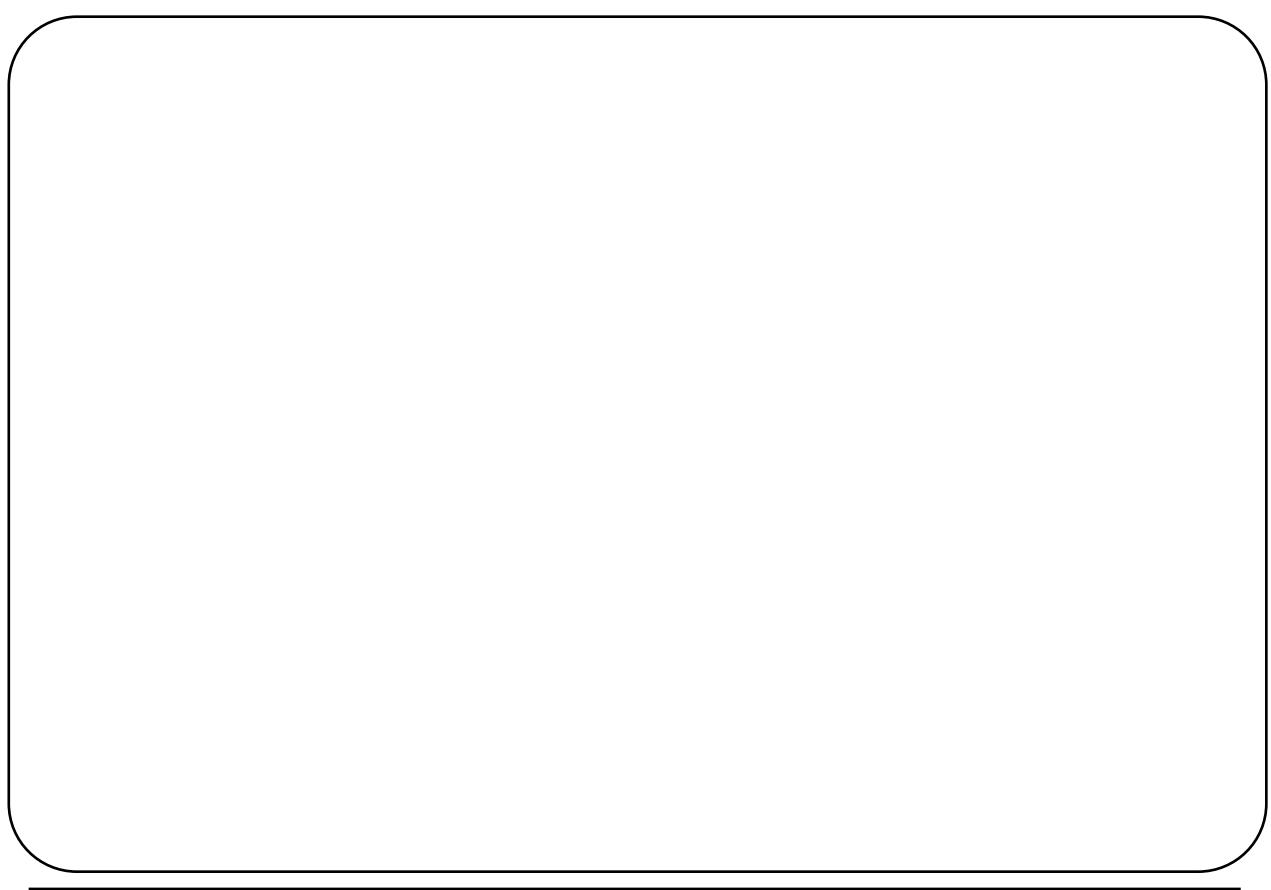
The example opposite shows how to cope with erroneous input. The main program loops round reading a double and writing it to the screen on each iteration until the end of file character is detected. Input errors are handled by discarding characters.

When using the `cin` object to perform input it is essential to check the status of the input buffer; several methods are provided for this purpose. Each of the following methods return true if the buffer is in the state indicated:

**good()**      buffer is OK  
**bad()**      buffer is not OK  
**fail()**      operation failed  
**eof()**      end of file detected

If an unexpected input character turns the stream's status to fail, it is essential to clear the stream and then discard the offending character. The following methods should be used:

**clear()**      clear stream's state  
**ignore(n, ch)**      ignore up to n characters (delimited by ch)



# 9

## Extensions to C

- **Function Name Overloading**
- **Heap Management**
- **Default Function Parameters**
- **Inline Functions**
- **Enumerated Types**



9

In this chapter we will take a look at several C++ extensions (improvements) to C.

C++ permits different functions to have the same name (**function name overloading**) provided they have distinct parameter lists. This lets you use the same names for functions that perform identical tasks on different data types.

C++ provides improved heap management facilities using the **new** and **delete** operators. These operators replace C's *malloc* and *free*.

Function parameters can now have default values. If a parameter's value is omitted from a call to the function then the default value is used instead.

In C, the preprocessor is used to provide **inline** functions (viz. #define with parameters). In C++, the compiler itself can generate inline functions.

**Enumerated types** are available in both C and C++. However, C++ has improved upon the C implementation and introduced *anonymous enumerated types*.

## Function Name Overloading

```
int Square(int n) {
    cout << "The square of integer " << n << " = " << n * n << endl;
    return n * n;
}
```

```
double Square(double n) {
    cout << "The square of double " << n << " = " << n * n << endl;
    return n * n;
}
```

```
void Square(char ch) {
    cout << "The square of character "
        << ch << " = "
        << ch << ch << endl;
}
```

```
int main()
{
    int n;
    double d;
    n = Square(5);
    d = Square(3.5);
    Square('A');
}
```

C++ allows the same function name to be multiply defined provided that each function has a unique signature (list of input parameter types). In this example the functions:

```
int Square (int)
double Square (double)
void Square (char)
```

all have different signatures and the compiler can therefore distinguish between the three calls in the main program:

<b>Square(5)</b>	<b>calls</b>	<b>int Square (int)</b>
<b>Square(3.5)</b>	<b>calls</b>	<b>double Square (double)</b>
<b>Square('A')</b>	<b>calls</b>	<b>void Square (char)</b>

Note that the return type is not considered as part of the function's signature. It is not permitted to define two functions differing only in return type, for example:

```
void Print(int);
int Print(int);
```

If this type of overloading were permitted the following code would be ambiguous:

```
Print(500);
```

Is this a call to void Print(int) or have we called int Print(int) and simply ignored the return code?

## Ambiguities

```
int main ()
{
    double m;
    m = Max(5, 12.667);
}
```

```
int Max(int a, int b) {
    if (a > b)
        return a;
    else
        return b;
}
```

```
double Max(double a, double b) {
    if (a > b)
        return a;
    else
        return b;
}
```

Function name overloading is an essential part of C++, but on occasions this feature introduces ambiguities into programs. Such ambiguities are nearly always the result of defining two or more functions with parameter lists that are almost identical.

Consider the example opposite. We have defined two Max functions: one function works with ints, the other with doubles. When Max is called

**Max(5, 12.667)**

it is called with parameters int and double. Since there is no Max function defined that takes mixed parameters the compiler attempts to use C style casting to obtain a match. Here, this introduces an ambiguity because the function

**int Max(int, int)**

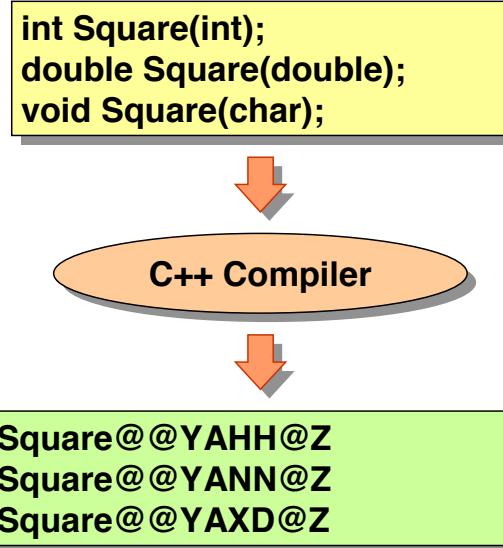
could be used if the second parameter (12.667) was cast to an int. But equally  
**double Max(double, double)**

could be used if the first parameter (5) was cast to a double.

Since the return type needs to be a double you might expect the compiler to give preference to the function using doubles. Furthermore, the function using ints truncates its second parameter as a result of the cast and hence it would be better to use the function using doubles. Nevertheless the compiler ignores all such considerations and regards both functions as equally attractive.

In the event of an ambiguous call, the compiler generates an error and you must recode your program. In this case you can simply remove the function that uses ints - it is superfluous. With only one function left to call, there are no ambiguities.

## Function Name Mangling



Copyright ©1994-2011 CRS Enterprises

118

Although the compiler can cope with different functions having the same name, the linker cannot. Therefore, the compiler translates function names (according to a compiler specific algorithm) into a new set of distinct names to keep the linker happy. This process is called name mangling. For example, the C++ might change the names as follows:

<b>int Square(int)</b>	<b>is transformed to ?Square@@YAH@Z</b>
<b>double Square(double)</b>	<b>is transformed to ?Square@@YANN@Z</b>
<b>void Square(char)</b>	<b>is transformed to ?Square@@YAXD@Z</b>

The mangling process is transparent to you the programmer unless you wish to write DLLs. You can inspect the compiler listing files to determine the mangled names.

Note that there is no standard for name mangling, each compiler vendor is free to choose their own algorithm. This makes it impossible to mix class libraries from different compiler vendors.

## Heap Management

**new  
new[ ]**

**delete  
delete[ ]**

```
int main()
{
    double* p;
    int* q;

    p = new double;
    q = new int [15];

    // use objects on heap

    delete p;
    delete [ ] q;
}
```

Copyright ©1994-2011 CRS Enterprises

119

C++ places great store in the heap! It is very common to create objects at run time in C++ (dynamic objects). The language provides two operators for heap management:

**new  
delete**

These operators should be used in preference to the malloc and free routines of C. The new operator (like malloc) returns a pointer to the area allocated on the heap.

To be more precise, C++ provides four heap management operators, not two:

<b>new</b>	<b>allocates objects on heap</b>
<b>new[ ]</b>	<b>allocates arrays of objects on heap</b>
<b>delete</b>	<b>deallocates objects on heap</b>
<b>delete[ ]</b>	<b>deallocates arrays of objects on heap</b>

Try not to mix them up, it can cause subtle bugs.

## new and delete

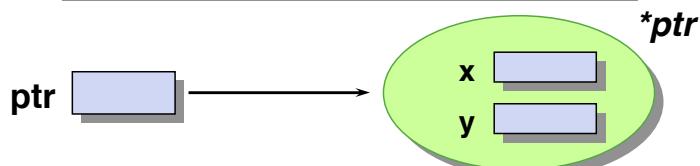
```
#include "Point.hpp"

int main()
{
    Point*     ptr;

    ptr = new Point;

    ptr->Set(4, 40);
    ptr->Move(3, 30);
    ptr->WhereAreYou();

    delete ptr;
}
```



Copyright ©1994-2011 CRS Enterprises

120

When an object is created dynamically it does not have a name. Therefore it is essential to use a pointer with a name to refer to the object. A local pointer can be used as in

**Point\* ptr;**

Once the pointer is established, the object can be created using

**ptr = new Point;**

Although the object does not have a name, we can refer to it as `*ptr` using standard pointer syntax.

When we need to send a message to the object we use the `->` notation as in  
**ptr->Set(4, 40);**

Remember that this is syntactically equivalent to

**(\*ptr).Set(4, 40);**

This later form is a clumsy but legal form. It illustrates that `*ptr` is an alias for our object.

All dynamic objects must be destroyed manually using the delete operator

**delete ptr;**

Failure to use delete will result in a memory leakage.

## new[ ] and delete[ ]

```
const int max = 10;

int main()
{
    int*      array;

    array = new int[max];

    for (int i = 0; i < max; i++)
    {
        array[i] = i * i;
        cout    << "Square of " << i << " is "
        << array[i] << endl;
    }
    delete [ ] array;
}
```

This example shows how to use the new [ ] and delete [ ] operators to create an array on the heap. The compiler allocates enough space for 10 ints; you do not have to use the sizeof operator.

We fill in the arrays with the squares of the first 10 integers. Note the declaration of i inside the for loop; i remains in scope for the remainder of the loop and cannot be referenced outside the loop (this is a recent change to the ANSI standard).

It is important to use delete [ ] (rather than delete without the brackets) to delete an array of objects referenced by the pointers array. Omitting the brackets is an error.

## Default Parameters

### *Total.hpp*

```
void total(int x1, int x2=10, int x3=100);
```

### *Total.cpp*

```
void total(int x1, int x2, int x3)
{
    cout << "Total is = "
        << x1 + x2 + x3
        << endl;
}

int main()
{
    total (5);
    total (5, 50);
    total (5, 50, 500);
}
```

A useful extension to C is the ability to give parameters default values when making a function call.

In this example the function total is called three times. The first call only supplies a value for x1 and therefore default values are used for x2 and x3. The second call uses a default value for x3 only. The last call supplies values for each parameter and therefore no defaults are used.

Note that default values for parameters must be specified in the appropriate header file, not in the source file.

## Inline Functions

### *Minmax.hpp*

```
inline int min (int x, int y) {
    if (x < y)
        return x;
    else
        return y;
}
inline int max (int x, int y) {
    if (x > y)
        return x;
    else
        return y;
}
```

### *Main.cpp*

```
#include MinMax.hpp

int main() {
    int x = 7, y = 10;
    cout << "Minimum is: "      << min(x, y)<< endl;
    cout << "Maximum is: "     << max(x, y)<< endl;
}
```

Copyright ©1994-2011 CRS Enterprises

123

In C++ you can have true inline functions; in C you had to use a #define statement in the preprocessor.

Using inline functions in C++ is far safer than using the preprocessor in C; inline functions do not suffer from the same notorious side effects that their C counterparts do. Note that the C++ compiler may refuse to expand the function inline if it is considered too large.

Obviously inline functions are expanded in situ making the executable code larger and the execution time faster; the stack is not used for the function parameters and return address.

Overuse of inline functions in large applications is discouraged; it can lead to excessively large executables being generated which cause unwanted paging and thus run very slowly.

## Inline Methods

### *Point.hpp*

```
class Point {
    // other definitions
public:
    void Set(int, int);
    void Move(int, int)
    {
        x = x + dx;
        y = y + dy;
    }
};
```

explicit inline

implicit inline

### *Point.cpp*

```
inline void Point::Set(int a, int b) {
    x = a;
    y = b;
}
```

explicit inline

Class methods can also be defined inline. However there are two different ways of defining inline methods.

We can define a method as inline using the same syntax as for a global function:

```
inline void Point::Set(int a, int b)
{
    x = a;
    y = b;
}
```

This is called an explicit inline method.

Alternatively we can define a method as inline by appending the code for the method to its function prototype inside the class definition:

```
class Point
{
    void Move(int, int)
    {
        x = x + dx;
        y = y + dy;
    }
};
```

This is called an implicit inline method.

The code for implicit and explicit inline methods must be placed in the class header file and not in the class implementation file. This is because the inline method definitions must be available to all compilation modules and not just to the class implementation file.

## Enumerated Types

```

enum Compass { North = 0,
                East = 90,
                South = 180,
                West = 270 };
enum Color { Red, Orange, Yellow, Green };
enum { Yes, No };

int main() {
    int answer;
    Color light;
    Compass direction;

    answer = Yes;
    if (answer == Yes)
    {
        direction = South;
        light = Yellow;
    }
}

```

Copyright ©1994-2011 CRS Enterprises

125

Enumerated types implement sets in C and in C++. In C each set must have a name; in C++ sets can be anonymous or be named.

The Compass directions are an example of a named set that consists of the items:

**North**  
**East**  
**South**  
**West**

and the named set Color consists of:

**Red**  
**Orange**  
**Yellow**  
**Green**

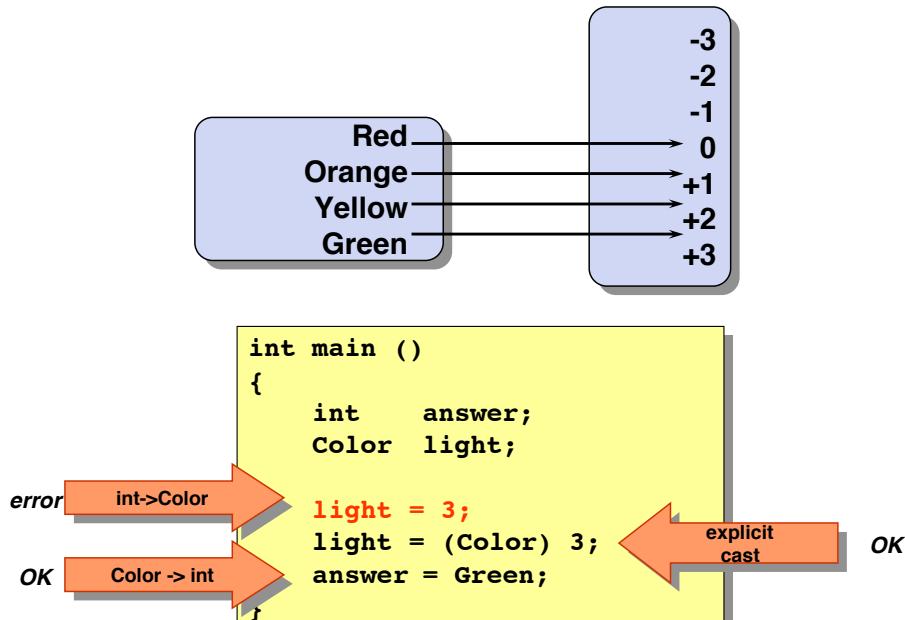
The set:

**Yes**  
**No**

is an example of an anonymous enumerated type.

Sets are implemented by mapping each item to a unique integer. You can specify the mapping (as shown in the enum Compass) or let the compiler choose. The compiler will always choose the mapping 0, 1, 2, 3 etc.

## Casting enums

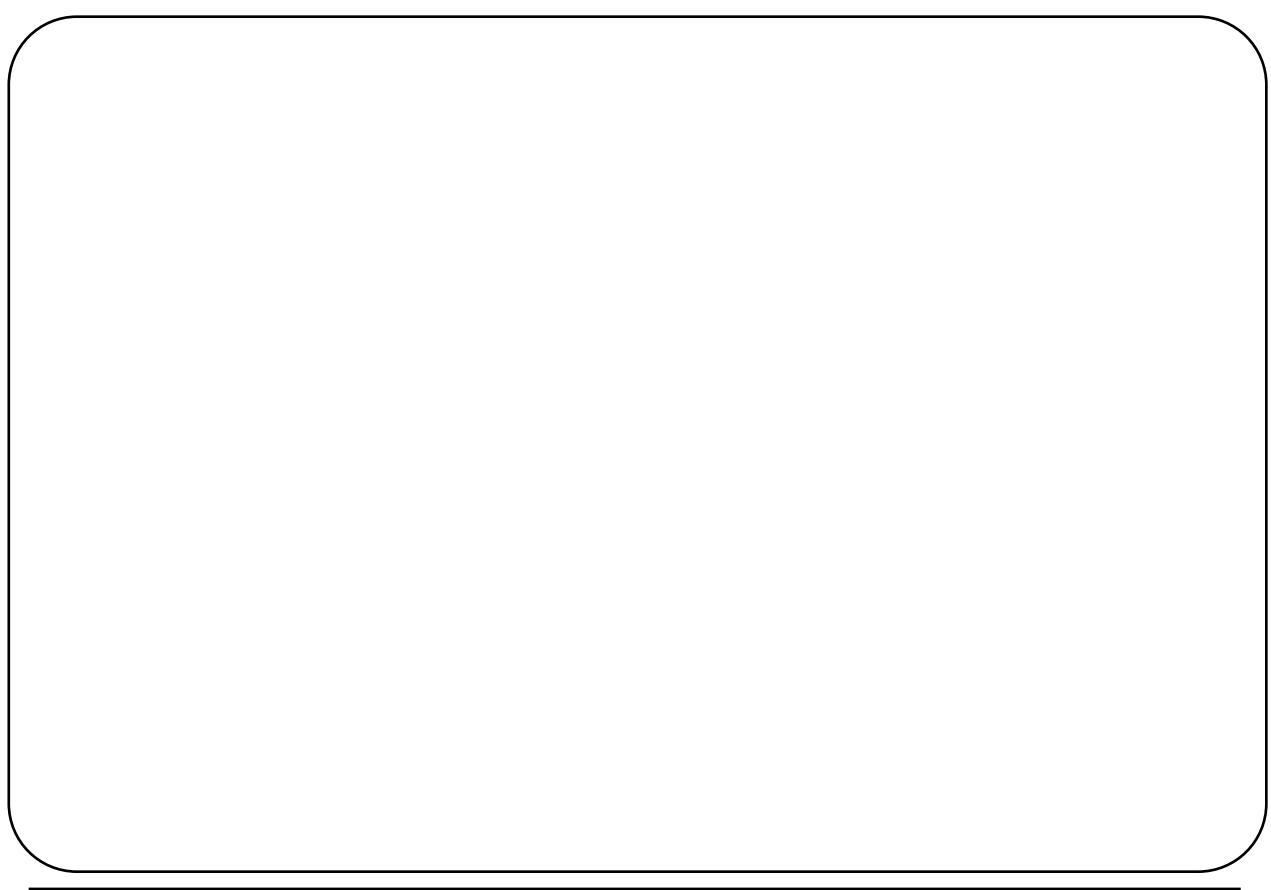


Copyright ©1994-2011 CRS Enterprises

126

In C it is possible to freely convert between enums and integers using a cast operator. However, most integers will not have a mapping defined for the enum and hence such conversions will be unsafe.

The C++ compiler prevents this happening by banning all implicit conversions from integers to enums. Conversions from enums to integers are always well defined and therefore permitted. Explicit casts may be used to force (potentially unsafe) conversions.



# 10

## References

- **References as Aliases**
- **Call by Reference**
- **Reference Returns**



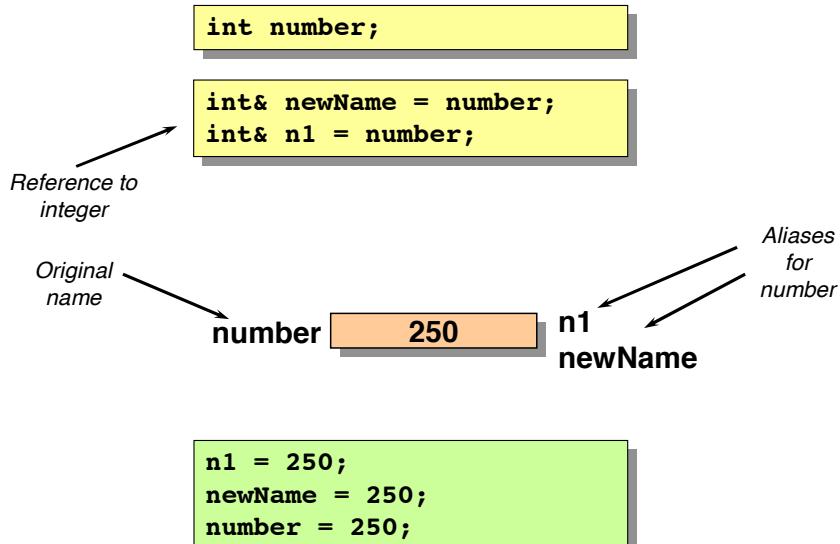
10

In this chapter we investigate C++ references. A reference is a simple data type that is less powerful but safer than the pointer type inherited from C. The name C++ reference may cause confusion, as in computer science a reference is a general concept data type, with pointers and C++ references being specific reference data type implementations.

References are used in three areas:

- **Providing alternative names (aliases) for existing variables**
- **Supporting function calls by reference**
- **Allowing functions to return references**

## References are Aliases



Copyright ©1994-2011 CRS Enterprises

130

References may be used to create a new name with which to reference an existing variable. The reference can be thought of as an alias for the original variable. References may be defined for class objects or built in data types. Declaring a reference does not allocate new storage - it merely provides another way of naming existing storage.

In this example, an integer variable **number** is declared and then a reference **newName** defined. The reference is declared using the notation:

**int&**

This should be read as a reference to an integer. The integer now has two names:

**number**  
**newName**

## Using Aliases

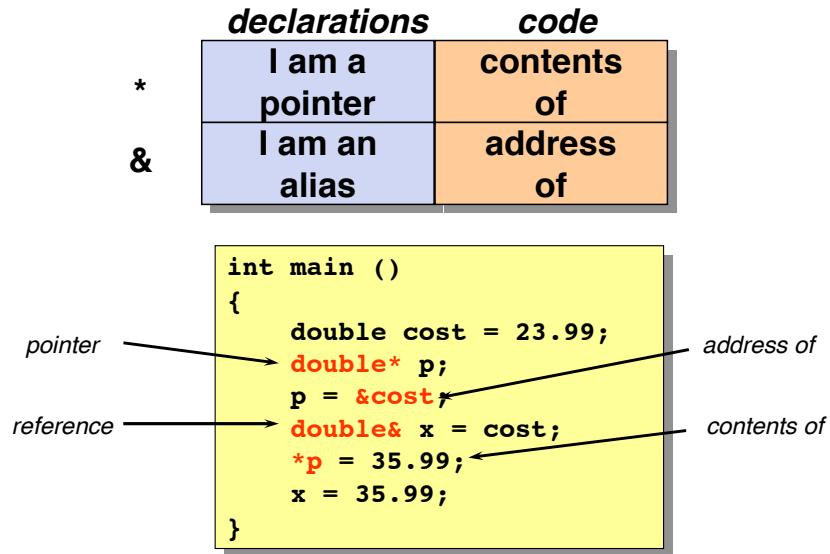
```
int main ()
{
    int array [3][2] = { {0,1}, {2,3}, {4,5} };
    int formula;
    int& x = array [1][1];
    int& y = array [2][0];

    formula = x * x * x + 2 * x * y - 7 * y;
}
```

References are often used to simplify names of variables. A typical case is shown opposite. A two dimensional array called array has been declared and initialized; we want to give new names to two elements of the array to simplify a complex formula.

The reference declarations set up aliases such that x can be used in place of the more cumbersome array[1][1] and y can be used in place of array[2][0].

## Operators \* and &



Copyright ©1994-2011 CRS Enterprises

132

The \* and & operators are used in C++ with multiple meanings. This can be extremely confusing to the beginner.

It is essential to distinguish between the use of these operators in declarations and in code. The operators have completely different meanings in these two contexts.

The \* operator causes a great deal of confusion in C, because in a declaration such as:

**int\* numberPtr;**

it means the numberPtr variable is a pointer, whereas in the code:

**n = \*numberPtr;**

the same operator refers to the contents of the location at the end of the pointer (i.e. not the pointer itself). These two meanings are almost diametrically opposed.

This tradition is not only maintained in C++, but extended by using the & operator to denote reference variables. In C++, the & operator has two completely different meanings, dependent on the context in which it is used.

## Call by Reference (in C)

```
void Swap(int* pa, int* pb)
{
    int temp;

    temp = *pa;
    *pa = *pb;
    *pb = temp;
}
```

```
int main()
{
    int x = 7;
    int y = 10;

    Swap (&x, &y);
}
```

In C, all function calls use the pass by value method of transferring parameters to a function. Pass by value places copies of the parameters on the stack and the called function works with these copies.

If you needed to work with the original data in C, instead of copies, then pointers had to be used (as in the example opposite). Working with original data is called call by reference.

## Call by Reference (in C++)

```
void Swap(int& a, int& b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;
}
```

```
int main ()
{
    int x = 7;
    int y = 10;

    Swap (x, y);
}
```

Copyright ©1994-2011 CRS Enterprises

134

In C++, you can pass parameters by value or by reference. There is no need to use pointers for call by reference because references can be used directly.

In the example opposite, the Swap function takes two reference parameters:

**Swap (int& a, int& b)**

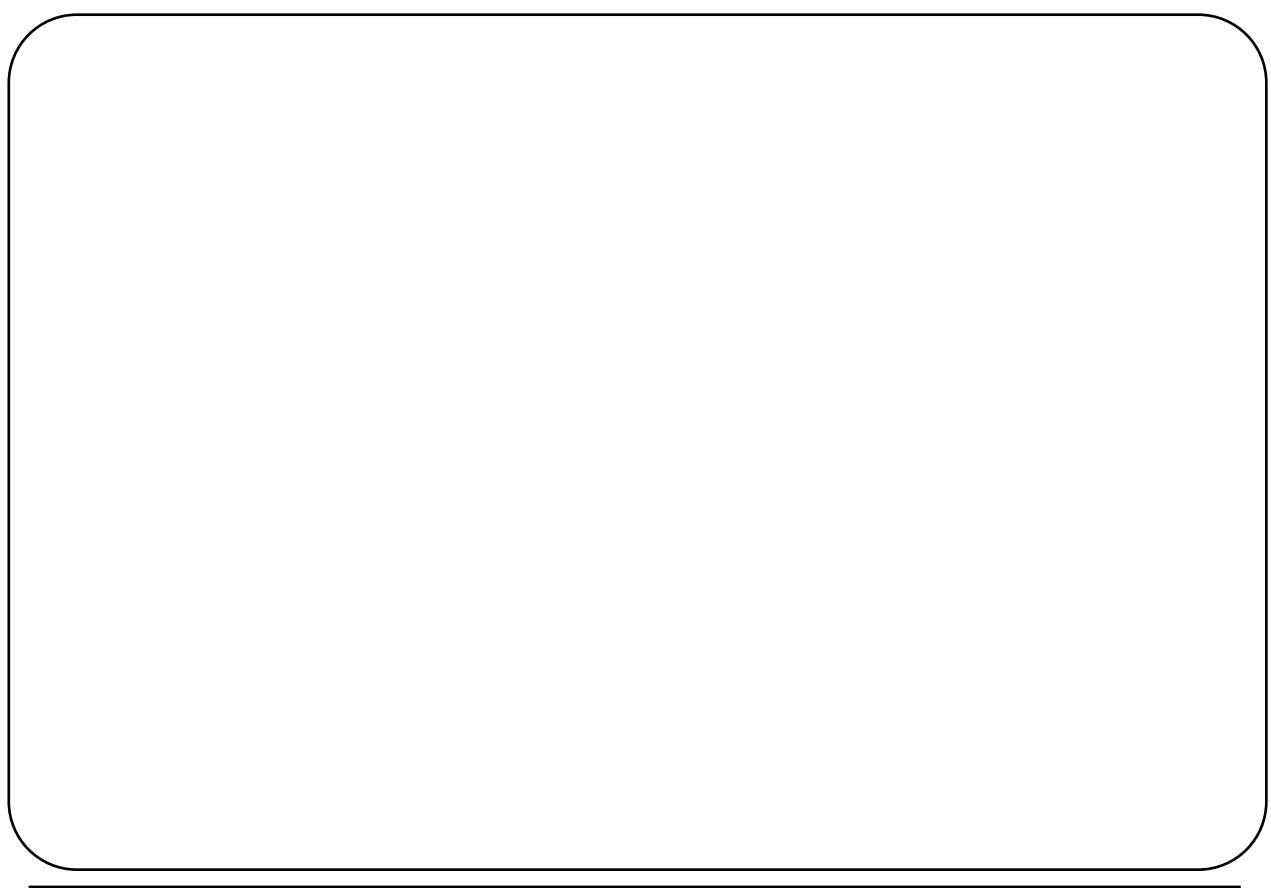
The references are bound to x and y of the main function by the call:

**Swap (x, y);**

This call has the semantics of initialization. It is as if we had said:

**int& a = x;**  
**int& b = y;**

Referring to a and b in the Swap function is therefore the same thing as using x and y directly. Of course, x and y are not in scope for Swap so the references must be used.



**11**

## Classes

- Constructors
- Destructors
- Initialization Lists
- Constant Members
- Arrays of Objects



11

Copyright ©1994-2011 CRS Enterprises

137

The C++ programming language allows programmers to separate program-specific data types through the use of classes. Instances of these data types are known as objects and can contain member variables, constants, member functions, and overloaded operators defined by the programmer. Syntactically, classes are extensions of the C struct, which cannot contain functions or overloaded operators.

In C++, a structure is a class defined with the struct keyword. Its members and base classes are public by default. A class defined with the class keyword has private members and base classes by default. This is the only difference between structs and classes in C++.

# Class Definitions

## *Point.hpp*

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

class Point
{
private:
    string name;
    int x;
    int y;
public:
    Point();           // default constructor
    Point (string, int, int); // constructor
    ~Point();         // destructor
};
```

The header file contains function prototypes for a class's methods. These functions represent the interface to the application program and should therefore be declared public. Note the special methods called Point; both these functions are constructors. Constructors always have the same name as their class. Also notice the method called ~Point; this is a destructor. Constructors are used to initialize objects; destructors are used to clean up as objects die.

Constructors and destructors are not allowed to return anything and must be declared with the return type omitted (not void). Multiple constructors are allowed (with different signatures), but only one destructor (with no parameters) may be defined per class.

By having two constructors, we have a choice of how to initialize new objects of the Point class. The constructor with no parameters:

**Point()**

is called the default constructor. The other constructor has no special name.

# Class Implementation

## *Point.cpp*

```
#include "Point.hpp"

Point::Point() {
    x = 0;
    y = 0;
    name = "";
}

Point::Point(string theName, int theX, int theY) {
    x = theX;
    y = theY;
    name = theName;
}

Point::~Point() {
    cout << "Object has died" << endl;
}
```

Copyright ©1994-2011 CRS Enterprises

139

The methods are placed in a separate source file. This file will need to include its associated header file to obtain all the class prototypes:

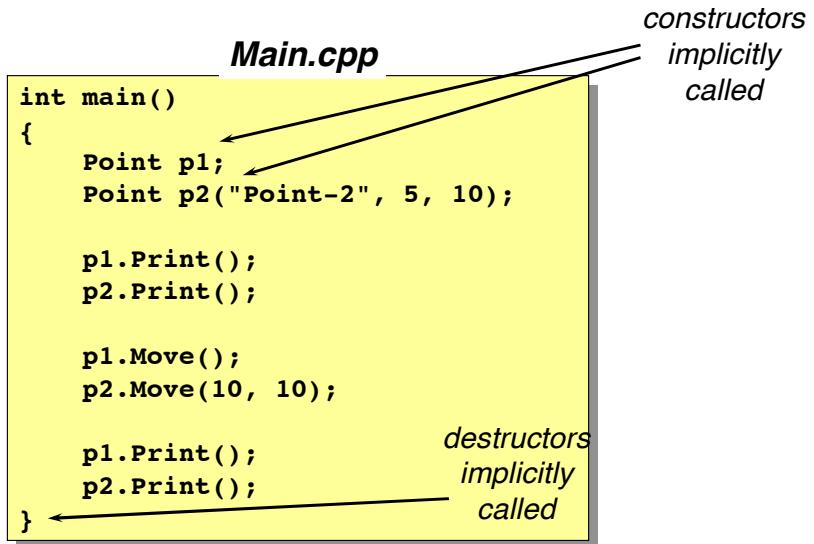
**#include "Point.hpp"**

Because it is possible to include methods from more than one class in the same source file, each method's name must be prefixed with the class name as in:

**Point::Point (constructor)**  
**Point::~Point (destructor)**

where :: is the scope resolution operator. Constructors are easy to spot because they have the same function name as the class.

## Invocation



The application program (often referred to as the client) should consist of easily read code. The complexity of an application should be wrapped up in the classes and not in the client code.

C++ makes a clear distinction between initialization and assignment. In C, the code fragment:

```

int x = 50;
int y;
y = 100;
  
```

initializes `x` (at compile time), but assigns to `y` (at run time). Unfortunately C uses the same operator for both initialization and assignment. C++ provides the initialization operator ( ) such that the above code fragment can be written:

```

int x(50);
int y;
y = 100;
  
```

The same method works for objects. The code fragment:

```

Point p2("point-2",5,10)
  
```

declares the object `p2` and then initializes it by implicitly calling the appropriate constructor. When these local objects go out of scope at the end of the function, the destructor is implicitly called.

## Initialization Lists

### *Point3D.hpp*

```
class Point3D
{
private:
    int x;
    int y;
    int z;
public:
    Point(int, int, int);
};
```

### *Point3D.cpp*

```
Point3D::Point3D(int theX, int theY, int theZ)
:
    x(theX),
    y(theY),
    z(theZ)
{ }
```

In the previous examples we have initialized objects using the function body of the constructor. However an alternative method of initialization is available: initialization lists.

Every constructor is defined as having an initialization part and an assignment part:

```
Point3D::Point3D(parameters)
:
    initialization-part
{
    assignment-part
}
```

It is recommended that all initialization is performed inside an initialization list and that the assignment part (body) is only used for any other processing required.

In the example shown opposite, all three data members of the Point3D object are initialized in the initialization list. The body of the constructor is empty! Constructors are often defined with an empty body.

## Initializing Arrays

```
class Polygon
{
private:
    int centreX;
    int centreY;
    int list[10];
public:
    Polygon(int, int, int[ ]);
};
```

**Polygon.hpp**

```
Polygon::Polygon(int x, int y, int theList[ ]
:
centreX(x) ,
centreY(y)
{
    for (int i = 0; i < 10; i++)
        list[i] = theList[i];
}
```

**Polygon.cpp**

You should always use the initialization list to initialize objects unless the object contains an array. Unfortunately, C++ does not support a syntax to initialize arrays in the initialization list, so arrays must be initialized in the body of a constructor.

In this example, we have initialized everything in the initialization list apart from the integer array. The array must be assigned in the body of the constructor:

```
for (int i = 0; i < 10; i++)
    list[i] = theList[i];
```

## Initializing Constants

### *Point.hpp*

```
class Point
{
private:
    const int id;           // read only
    int x;
    int y;
public:
    Point (int theId, int theX, int theY);
};
```

### *Point.cpp*

```
Point::Point(int theId, int x, int y) :
    id(theId),
    x(theX),
    y(theY)
{}
```

Classes sometimes have constant data members. The compiler will allow constants to be initialized, but not assigned. This means that constant data members have to be part of the initialization list. Assignment to a constant in the body of a constructor is illegal.

The x and y attributes of the class are variables and can be initialized as shown or as part of the constructor body. It is better style to initialize everything in the initialization list, but you are permitted to write

```
Point::Point(int theId, int theX, int theY) :
    id(theId)
{
    x = theX;
    y = theY;
}
```

if you prefer.

## Initialization Summary

### *Point.cpp*

```
Point::Point(parameters)
:
// initialization list
{
// constructor body
}
```

**initialization list**  
**variables**  
**constants**  
**references**  
**aggregation**  
**inheritance**

**constructor body**  
**variables**  
**arrays**

Remember, always initialize objects in the constructor initialization list and only use the body of the constructor for arrays. Note that for most types of C++ data, it is mandatory to use the initialization list.

With variable data, you have a choice. It is legal to initialize variables in the initialization list or in the constructor body. The choice is a matter of style. Arrays must be initialized in the constructor body. Everything else must be initialized in the initialization list.

## Which Constructor?

```
class Point
{
public:
    Point();
    Point(int, int);
    Point(const string&);

private:
    int x;
    int y;
};
```

**Point.cpp**

```
int main()
{
    Point p1(4, 7);
    Point p2;
    Point p3("6:8");
    Point p4;
}
```

**Main.cpp**

Copyright ©1994-2011 CRS Enterprises

145

Most classes have several constructors. This enables objects to be initialized in a variety of ways. In this example we have 3 different constructors:

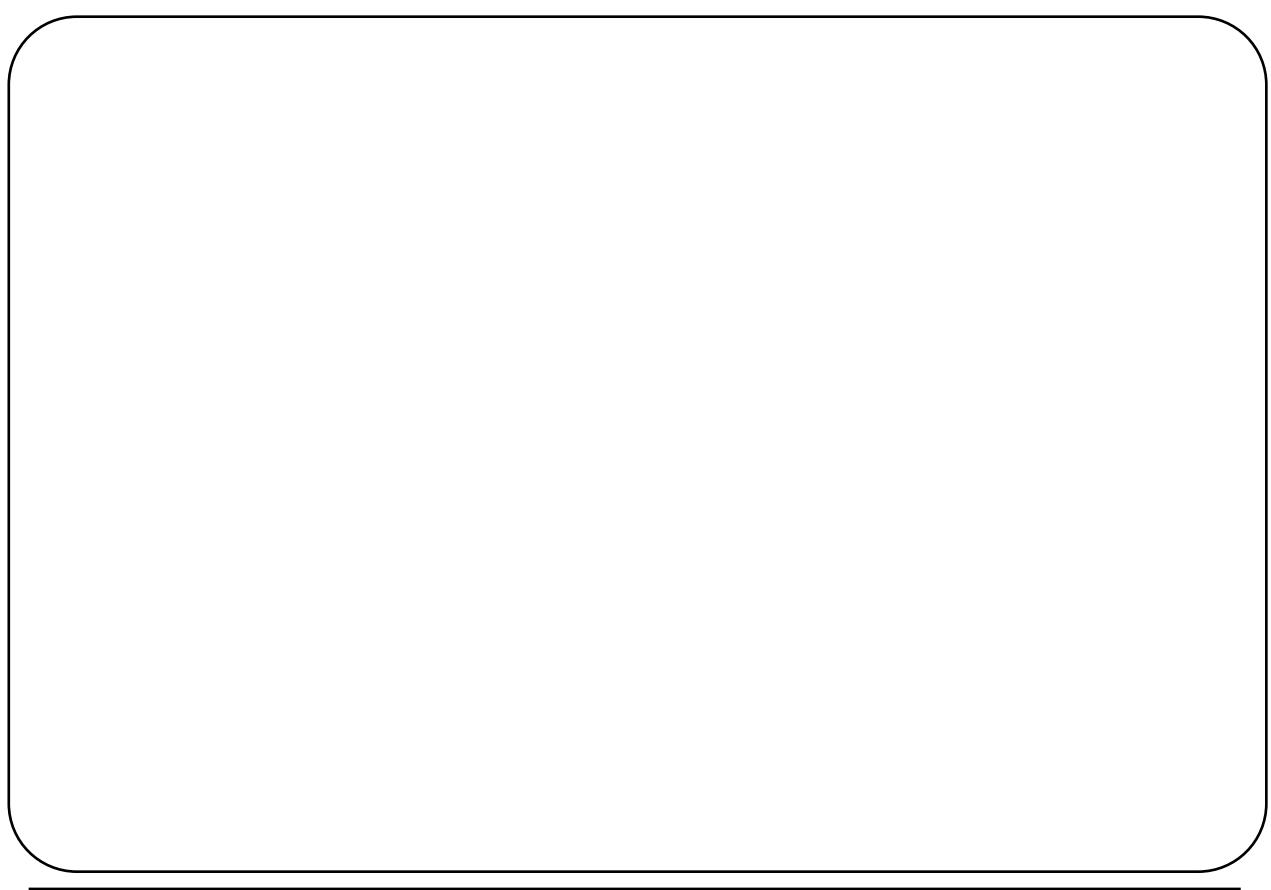
```
Point();           // default constructor
Point(int, int, int);
Point(char*);
```

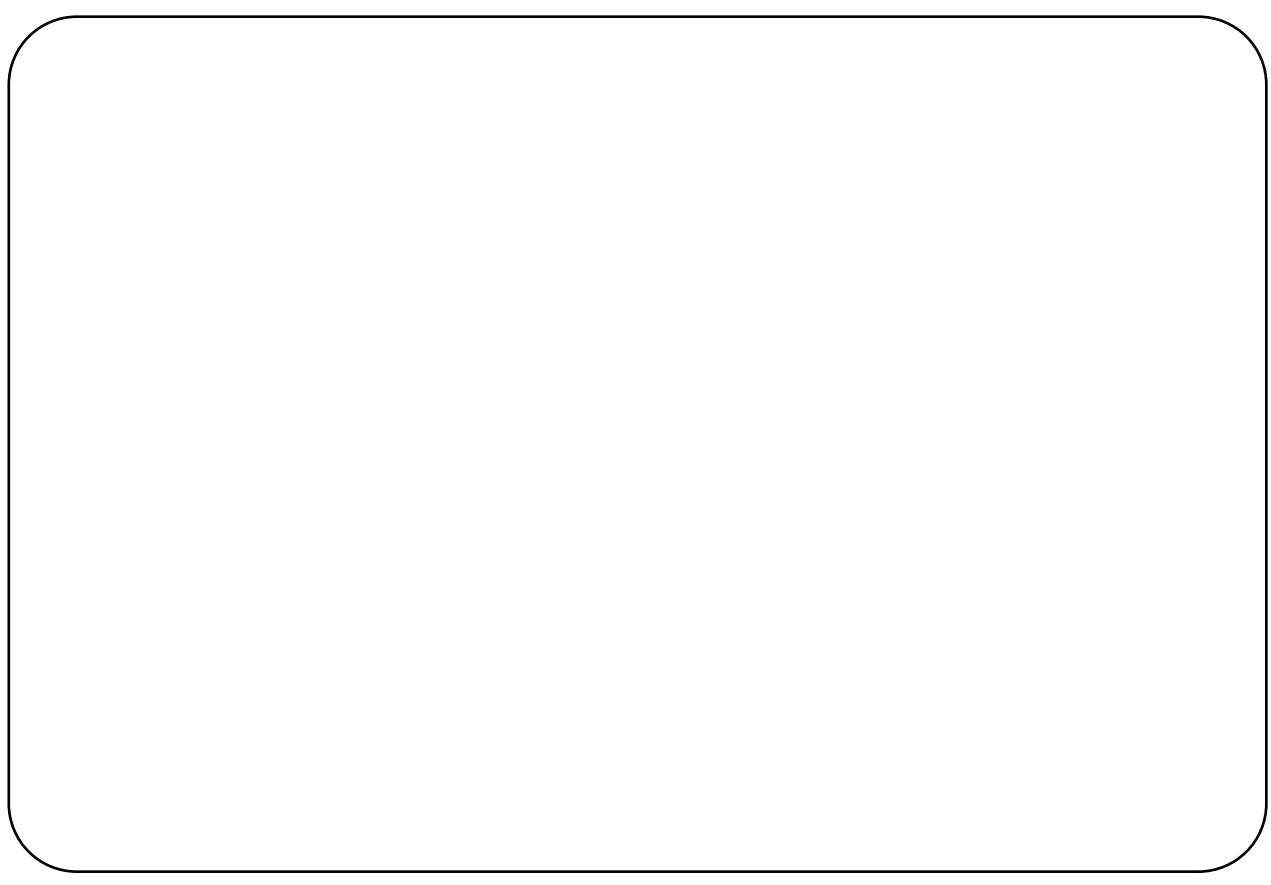
that are used to initialize 4 objects: p1, p2, p3 and p4.

The default constructor is used to initialize p2 and p4.

The Point(int, int, int) constructor is used to initialize p1.

The Point(string) constructor is used to initialize p3. Using a string to initialize a Point object is somewhat offbeat. We show it here to illustrate that the constructor parameter list can be quite dissimilar to the attributes of the object being initialized.





# 12

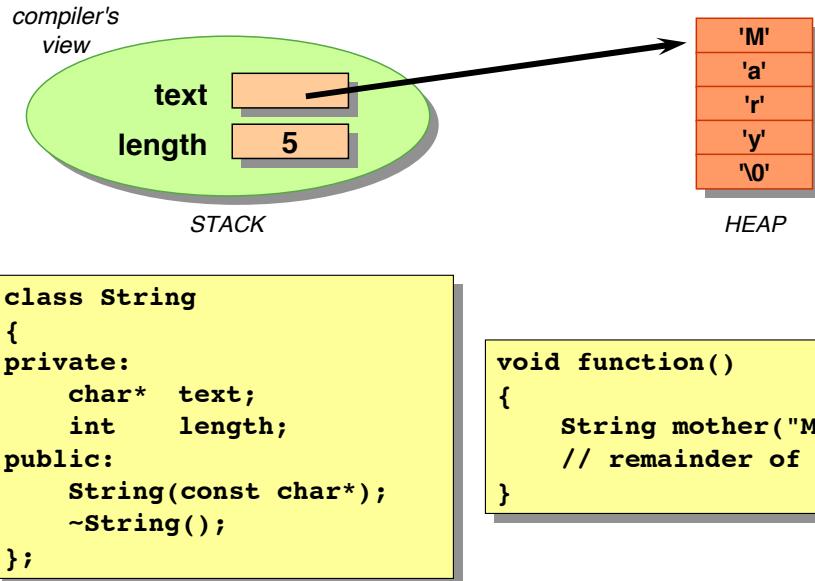
## More on Classes

- Variable sized Objects
- Constant Objects
- Statics



12

## Variable Sized Objects



Copyright ©1994-2011 CRS Enterprises

150

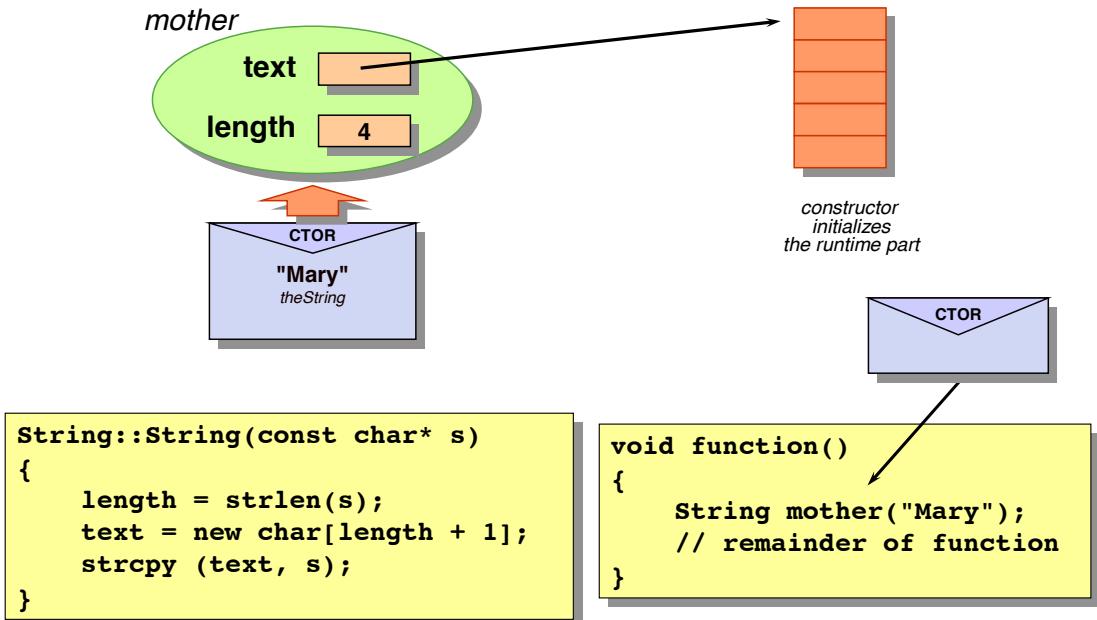
It is very common in C++ to build objects that refer to data on the heap. We embed pointers in objects to point at this information. Such objects can be thought of as incomplete: the compiler is only aware of the class definition of an object and has no way of knowing about this additional information.

Constructors are used to allocate and initialize the additional storage for incomplete objects. The destructor is used to deallocate the storage allocated within the constructor.

Care must be exercised when an incomplete object is destroyed. Destruction takes place when the variable goes out of scope (for local and global objects) or when delete is called (for dynamic objects). The compiler will arrange for automatic deallocation of an object, but will not deallocate any associated data on the heap. You must write a destructor to deallocate this storage.

In this chapter we will define our own class called `String` which exhibits this behavior. This class is analogous to the `string` class defined in the standard template library.

## Calling the Constructor



Copyright ©1994-2011 CRS Enterprises

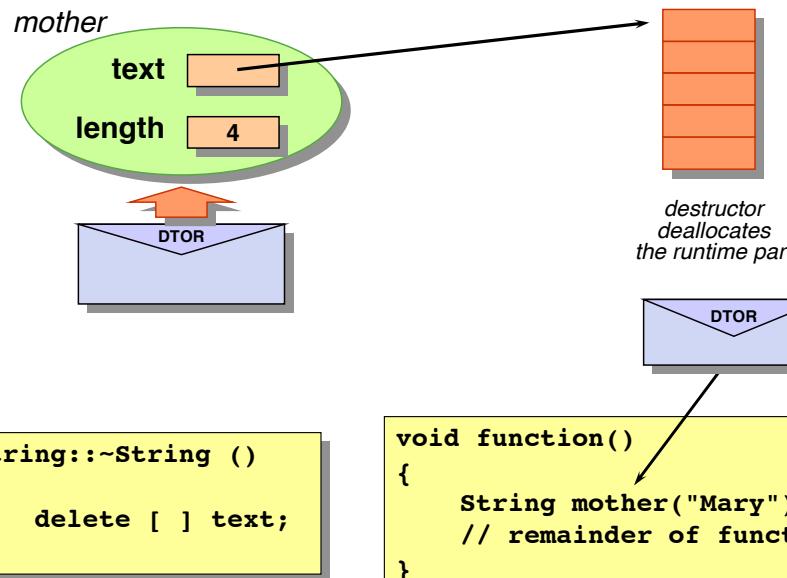
151

When a `String` object is created the constructor is automatically called. The constructor's primary concern is to initialize length and text of the object. The constructor's secondary concern is to allocate heap space to store a copy of the string.

Notice that we used the body of the constructor to initialize length and text (i.e. assignment) rather than using the initialization list. Indeed, length could have been placed in the initialization list, but text uses `new [ ]` which has to be performed at run time (hence must be assigned in the constructor body). If we used an initialization list our constructor would read:

```
String::String (const char* s)
:
length(strlen(s))
{
    text = new char[length + 1];
    strcpy (text, s);
}
```

## Calling the Destructor



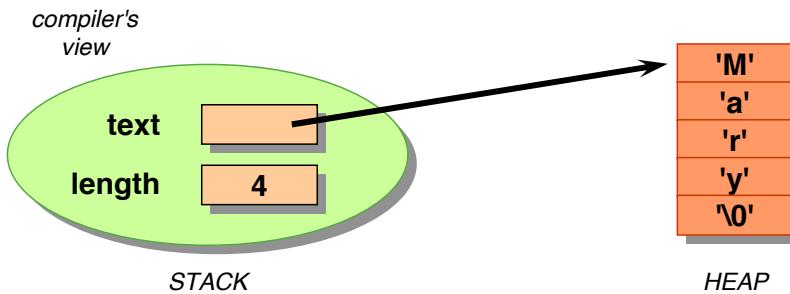
Copyright ©1994-2011 CRS Enterprises

152

For classes defining incomplete objects it is essential to provide a destructor member function. The destructor will be automatically called at the moment when an object is about to be destroyed (i.e. its memory deallocated). The destructor must arrange for deallocation of any associated data on the heap.

The destructor calls `delete` to deallocate the heap storage associated with the `String` object. The destructor does not need to worry about the stack part of the object (the compiler deallocates that); the destructor's job is to deallocate any storage allocated in the constructor.

## Construction Summary

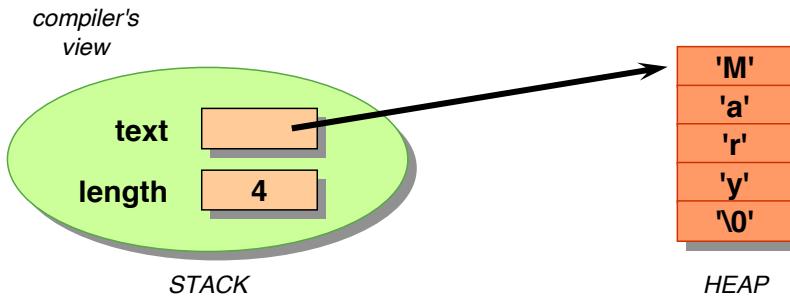


1. **Compiler allocates storage on STACK**
2. **Compiler calls constructor**
3. **Constructor initializes STACK**
4. **Constructor calls new[ ]**
5. **new [ ] allocates space on HEAP**
6. **constructor initializes HEAP**

The steps involved in creating a String object are as follows:

1. The compiler allocates storage for the text and length attributes of the String object on the stack.
2. The compiler automatically calls the String constructor to initialize the String object.
3. The String constructor initializes the text and length attributes.
4. The String constructor calls new [ ].
5. The new [ ] operator allocates storage for the character array on the heap, but leaves the storage uninitialized. new[ ] then returns control to the constructor.
6. Finally the String constructor calls strcpy to initialize the storage on the heap.

## Destruction Summary

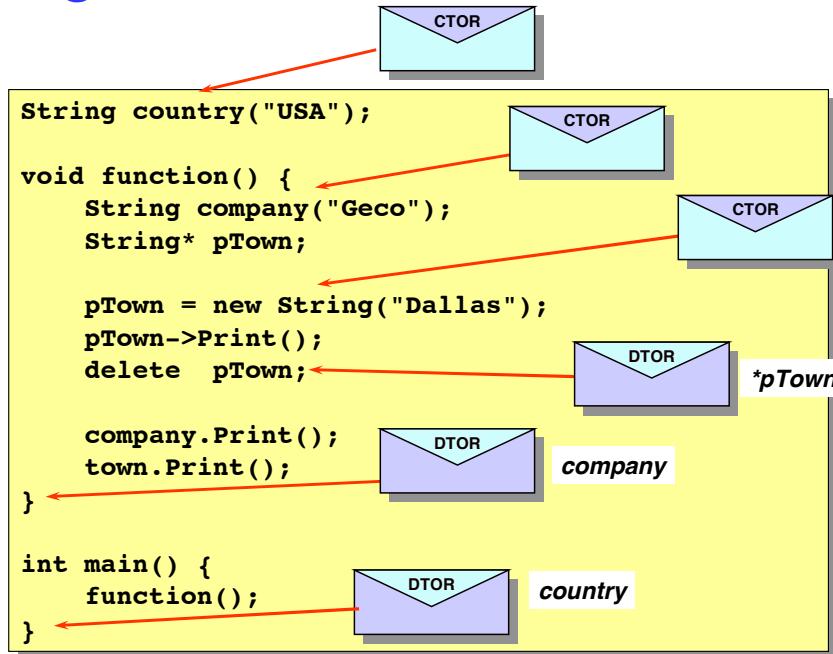


1. **Compiler calls destructor**
2. **Destructor calls delete[ ]**
3. **delete [ ] deallocated space on HEAP**
4. **Compiler deallocates storage on STACK**

The steps involved in destroying the incomplete String object are:

1. The compiler automatically calls the String destructor.
2. The String destructor calls delete [ ].
3. The delete [ ] operator deallocates storage for the character array and returns control to the destructor.
4. The compiler deallocates stack storage for the String object.

## Timing



Copyright ©1994-2011 CRS Enterprises

155

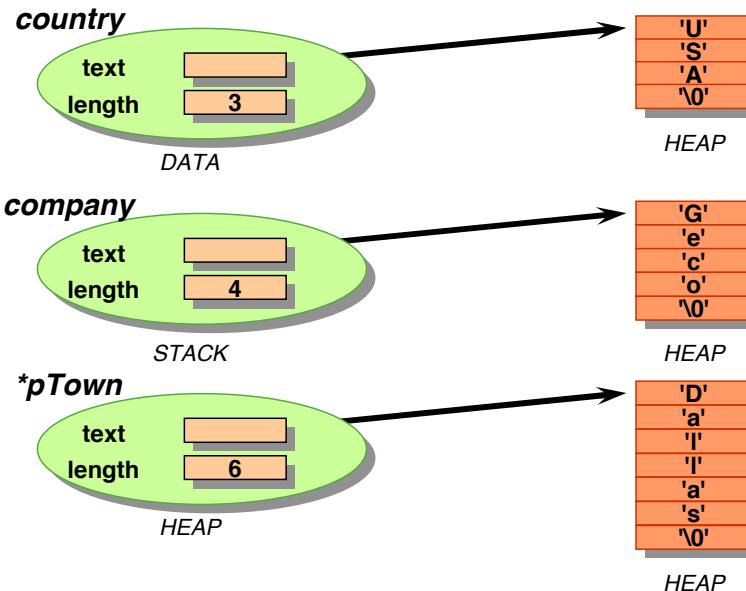
Note the timings of when constructors and destructors are called.

Constructors are called immediately after storage is allocated for the object (i.e. at the time they are declared). For global objects this means at the start of the program, for local objects the constructor is called at declaration time and for dynamic objects the constructor is called by `new`.

Destructors are called immediately before storage is deallocated for the object (i.e. at the curly brace defining the end of a function).

For global objects this means at the end of the program, for local objects the destructor is called at the curly brace defining the end of the function in which it was declared and for dynamic objects the destructor is called by `delete`.

## Storage



Copyright ©1994-2011 CRS Enterprises

156

It is instructive to compare creating incomplete objects on the stack, heap and data segments.

In all three cases, each object stores its data on the heap. The objects on the stack and data segments have names, but the object on the heap does not have a name; it has to be referenced via a pointer that does have a name.

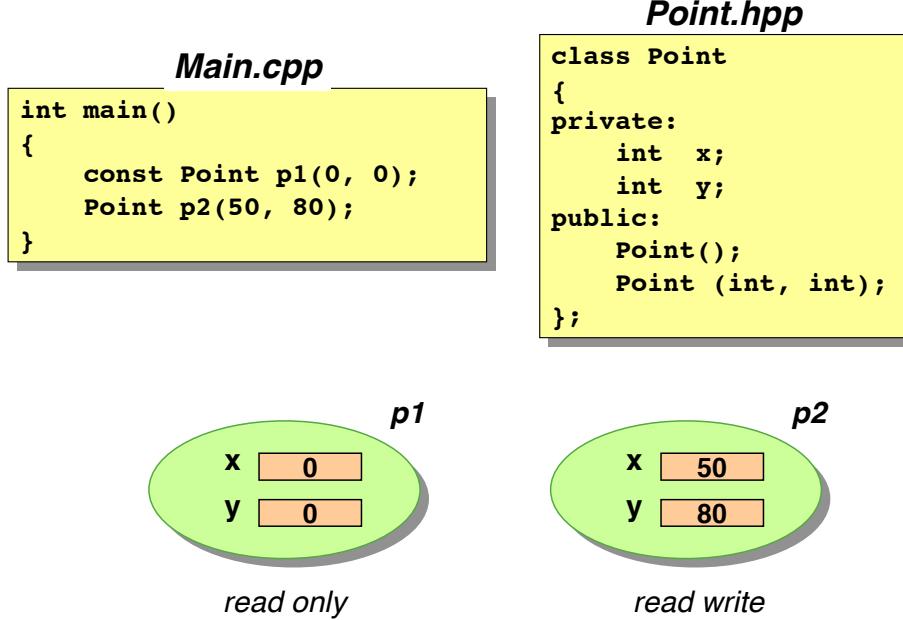
This means that to create an object on the heap, you must first create a pointer elsewhere. Hence, creating objects on the heap is more complicated than creating objects on the stack and data segments.

Creating objects on the stack and data segments is straightforward since all the hard work is performed by the compiler. Note that the compiler calls your constructors automatically and arranges for the destructor to be called as each object dies.

Creating objects on the heap requires using the `new` operator with the pointer you created earlier. When you have finished with the object you must use `delete` to clean up the heap.

Although creating objects on the heap is more complicated, you have complete control over when the object is created and when it is destroyed. Note that `new` and `delete` do not necessarily have to be used in the same function. You can create an object in one function and not delete it until much later (in another function) if you like.

## Constant Objects



Copyright ©1994-2011 CRS Enterprises

157

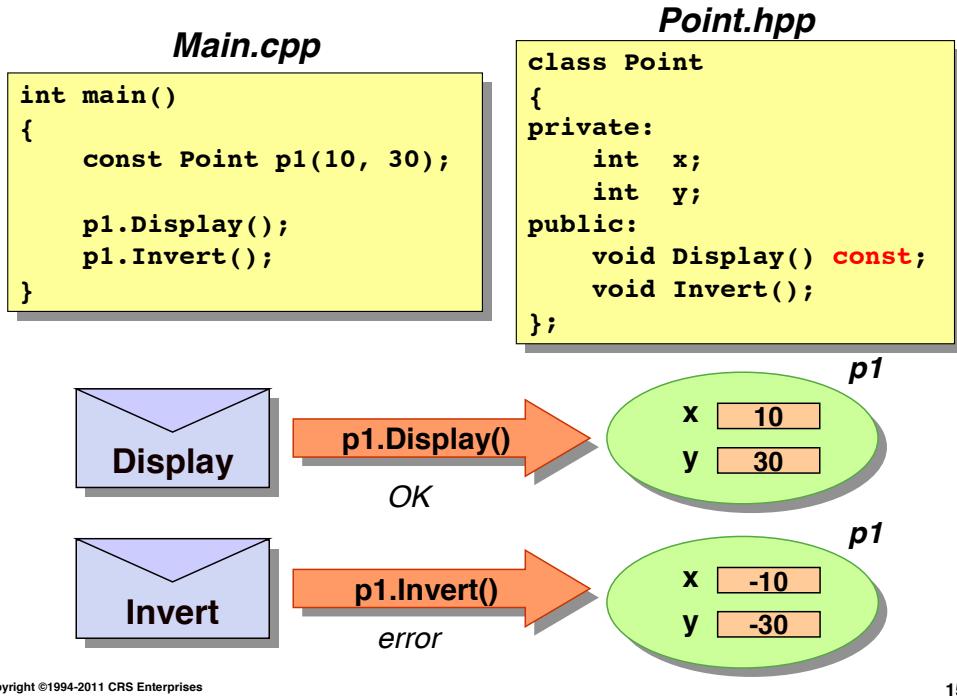
Sometimes you want an entire object to be constant (read only). This is achieved by declaring the object with the `const` attribute as in

`const Point p1(10, 30);`

and the appropriate constructor is called to initialize the object. Thereafter we are not permitted to forward any messages that attempt to modify the object.

This leads to the concept of two types of messages: constant (query) messages that do not attempt to change the object and non-constant messages that do change the object.

## Constant Methods



In this example **Display** is a constant message since it does not attempt to change **p1**. On the other hand, **Invert** is a non constant message and it illegal to send this message to **p1**.

Constant methods must be declared as such in the class definition. Note the positioning of the **const** keyword in

**void Display() const;**

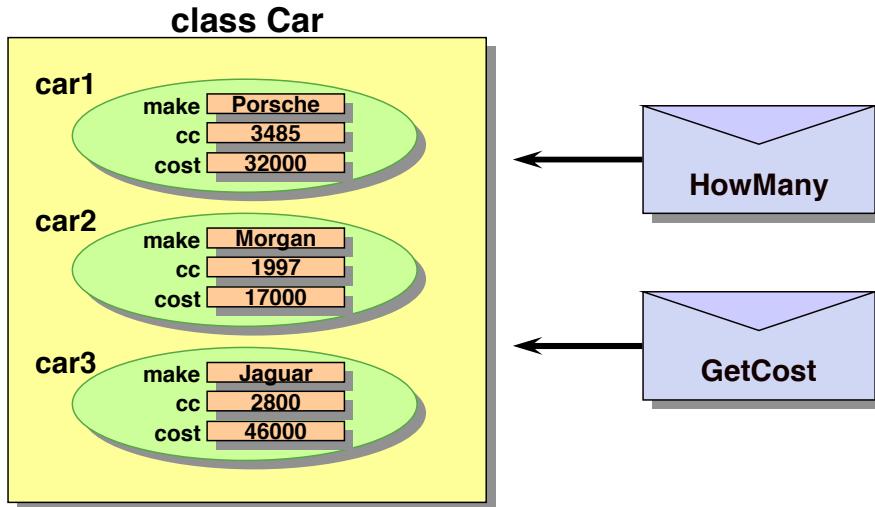
Far from elegant, but where else can we put the keyword? Inside the brackets implies input parameters are constant and at the beginning of the line implies the return is constant.

If we have two objects with one constant and two methods with one constant, four possibilities arise

object	method	valid?
const	const	OK
const	non-const	Error
non-const	const	OK
non-const	non-const	OK

Note that the compiler assumes a method changes an object unless you declare the method as constant. Obviously, constant methods are not allowed to call other non constant methods, that would invalidate the whole scheme.

## Shared Information



Copyright ©1994-2011 CRS Enterprises

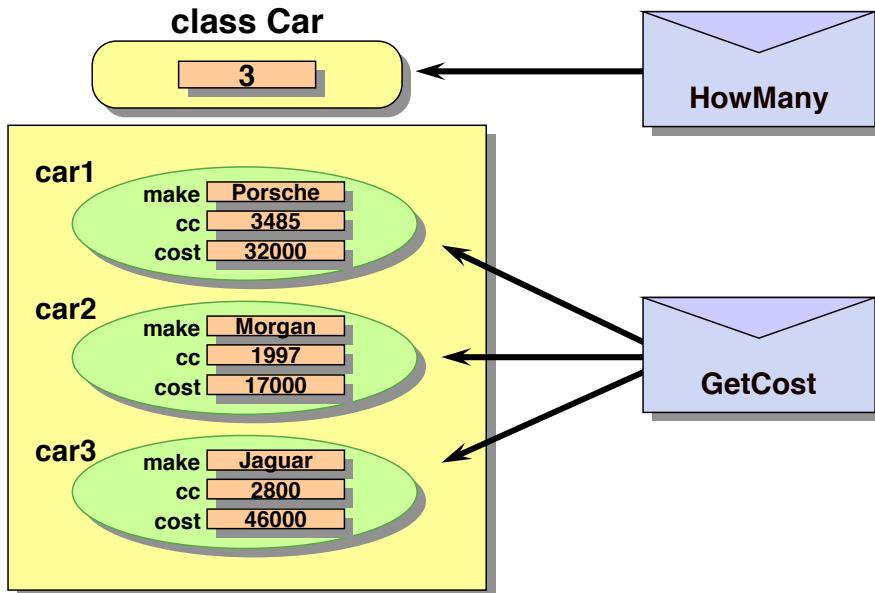
159

Consider a class **Car** that has 3 objects defined: **car1**, **car2** and **car3**. How can the client code determine that there are currently 3 objects defined for this class?

Obviously, the client code should send a **HowMany** message. But which object should the message be sent to? And what happens when no objects exist?

A normal message such as **GetCost** must be sent to the appropriate object since the cost of each object is different, but it appears that the **HowMany** message can be sent to any of the objects.

## The Class Object



Copyright ©1994-2011 CRS Enterprises

160

To solve our problem, C++ defines a special type of object called the class object. This object always exists, even before any real objects are created. The **HowMany** message can be sent to this object to determine the number of cars in existence. Once a car has been created it too can receive **HowMany** messages, but it is better styling to send the **HowMany** message to the class object.

## Static Modifier

```
class Car
{
private:
    static int count;
    string make;
    int c;
    int cost;
public:
    static int HowMany();
    int GetCost();
};
```

```
int Car::count = 0;

int Car::HowMany()
{
    return count;
}

int Car::GetCost()
{
    // implementation
}
```

The static modifier is used to mark class attributes as shared by the class. When applied to a method, the static modifier restricts the method such that it only has access to static attributes. The `HowMany` method is an example of a static method.

Shared attributes are not stored within each object of the class. Clearly this would duplicate shared information and be grossly inefficient. Shared attributes are stored within the class object. The storage for the class object is allocated by the syntax

`int Car::count = 0;`

This declares a static attribute and initializes it.

## Manipulating Statics ...

```
Car::Car()
:
    make  (theMake),
    cc    (theCc),
    cost   (theCost)
{
    count++;
}

Car::~Car()
{
    count--;
}
```

```
int main()
{
    Car car1("Porsche", 3485, 32000);
    Car car2("Morgan", 1997, 17000);
    Car car3("Jaguar", 2800, 46000);

    cout << Car::HowMany() << endl;
    cout << car1.GetCost() << endl;
}
```

Copyright ©1994-2011 CRS Enterprises

162

Static member functions are invoked by prefixing the message with the class name as in  
**Car::HowMany()**

Static methods have access to static data only. It is illegal to access non static data from a static method.

All that remains is to find suitable methods in which to increment and decrement the count of cars. The constructor and destructor are the obvious candidates.

## ... Manipulating Statics

```
Car::HowMany()
{
    cout << "Object at address: "
        << this << endl;
    cout << "Cost: " << cost << endl;

    return count;
}
```

```
int main()
{
    Car car1("Porsche", 3485, 32000);
    Car car2("Morgan", 1997, 17000);
    Car car3("Jaguar", 2800, 46000);

    cout << Car::HowMany()
        << car1.HowMany()
        << car2.HowMany()
        << car3.HowMany() << endl;
}
```

Copyright ©1994-2011 CRS Enterprises

163

Static methods differ from normal methods in a number of ways. We have seen that they can communicate with the class object and are only allowed to handle static data. Another very important point is that there is no this parameter defined for a static method. The this parameter exists for normal methods to identify the address of the object receiving the message. But the class object is not a real object and therefore has no address.

The code fragment

```
cout << "Object at address: " << this << endl;
```

fails to compile because of the attempted reference to this.

The code fragment

```
cout << "Cost: " << cost << endl;
```

also fails to compile; this time because cost is not static.

However the code fragment

```
return count;
```

is OK; count is static.

## Creating Namespaces

```
namespace Library1 {
    class List{
        public:
            void Print(){}
    };
    class Queue {};
    class Stack {};
}
namespace Library2 {
    class List{
        public:
            void Print(){}
    };
    class Queue {};
    class Stack {};
}
```

```
int main()
{
    Library1::List list1;
    list1.Print();

    Library2::List list2;
    list2.Print();
}
```

With the explosion of available third party class libraries, has come the growing problem of namespace pollution. It is more than likely that if you use more than one class library in a software system you will encounter name clashes between unrelated classes. Namespaces have been introduced to solve the problem of name clashes.

The namespace directive classifies code as belonging to a given namespace. When you wish to access any classes or functions that clash with those in other libraries, you must specify a namespace such as

**Library1::List list1;**

This resolves the ambiguous reference to List in the two libraries.

## Using Namespaces

```

int main()
{
    using namespace Library1;
    List list1;
    list1.Print();

    Library2::List list2;
    list2.Print();
}

```

*Library1 is the default*

*Library2 is not the default*

A nice feature of namespaces is the ability to define a default namespace. The statement **using namespace Library1**

means that the compiler will resolve ambiguous references in favour of the Library1 namespace. Note you can still access classes and functions in other namespaces but these names must be fully qualified.

Note that it is permissible to define more than one default namespace as in  
**using namespace Library1**  
**using namespace Library2**

Be careful, an unqualified reference to List is now ambiguous; we are back to where we started!

## Nested Classes ...

```

class List
{
    class Node
    {
        public:
            Node(double d)
            {
                pData = new double(d);
                pNext = 0;
            }
        private:
            Node* pNext;
            double* pData;
    };

    public:
        List(int theSize) : size(theSize) {}
    private:
        Node* p;
        int size;
};

```

```

int main()
{
    List theList(5);
}

```

Copyright ©1994-2011 CRS Enterprises

166

Nested classes allow you to encapsulate one class definition within another. This is handy if a class requires its own private data types. Originally C++ only supported this feature when the encapsulated class was defined physically inside its owner. The encapsulated class cannot be used outside its container class.

Member functions of the encapsulated class can be defined inline (as shown) or using double class scope resolution:

```

List::Node::Node(double d)
{
    pData = new double(d);
    pNext = 0;
}

```

## ... Nested Classes

```
class List
{
    class Node; // forward declaration
public:
    List(int theSize):size(theSize)
    {}
private:
    Node* p;
    int size;
};
```

```
class List::Node
{
public:
    Node(double d)
    {
        pData = new double(d);
        pNext = 0;
    }
private:
    Node* pNext;
    double* pData;
};
```

```
int main()
{
    List theList(5);
}
```

Copyright ©1994-2011 CRS Enterprises

167

Having to define a nested class physically inside its container class can lead to cryptic class definitions. The alternative syntax is now permitted and preferred.

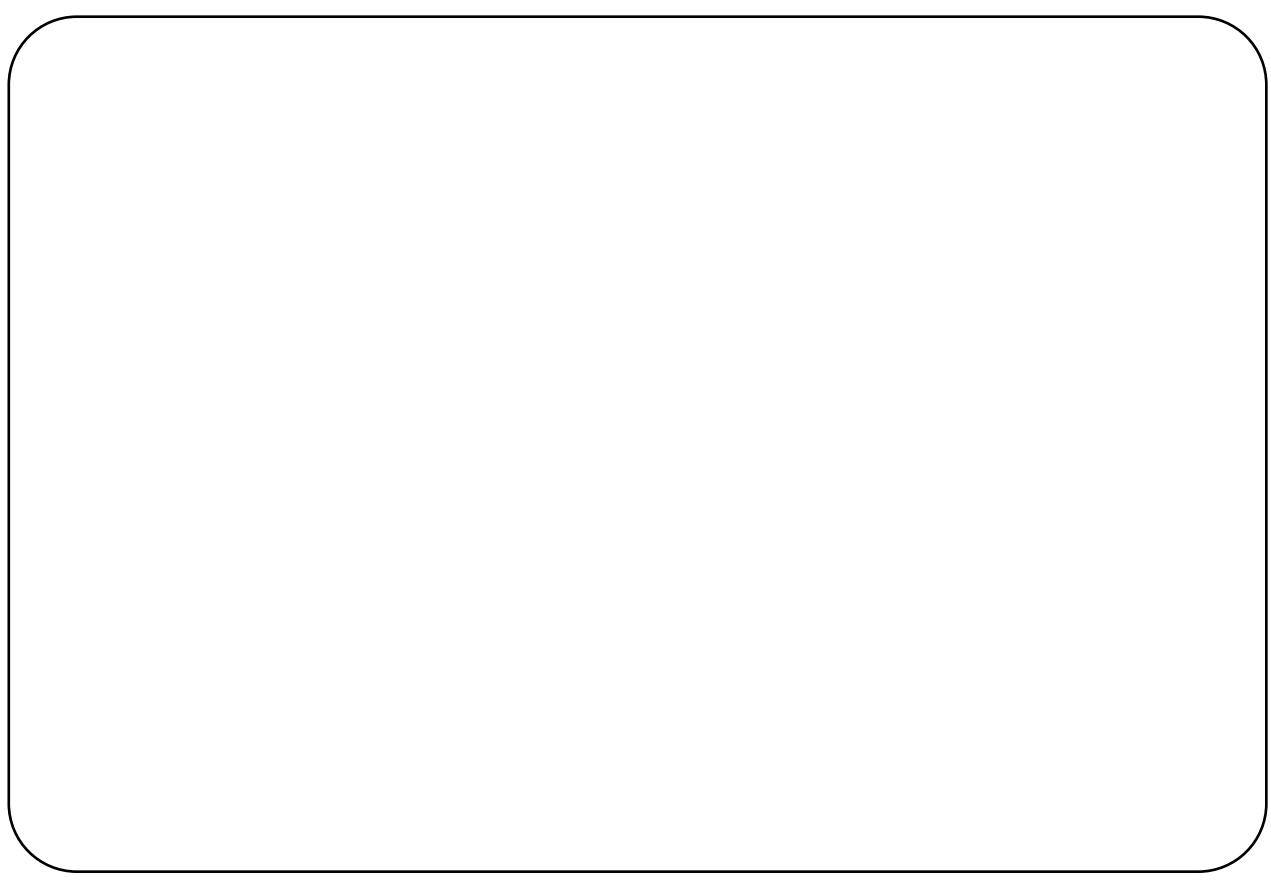
Note the forward declaration

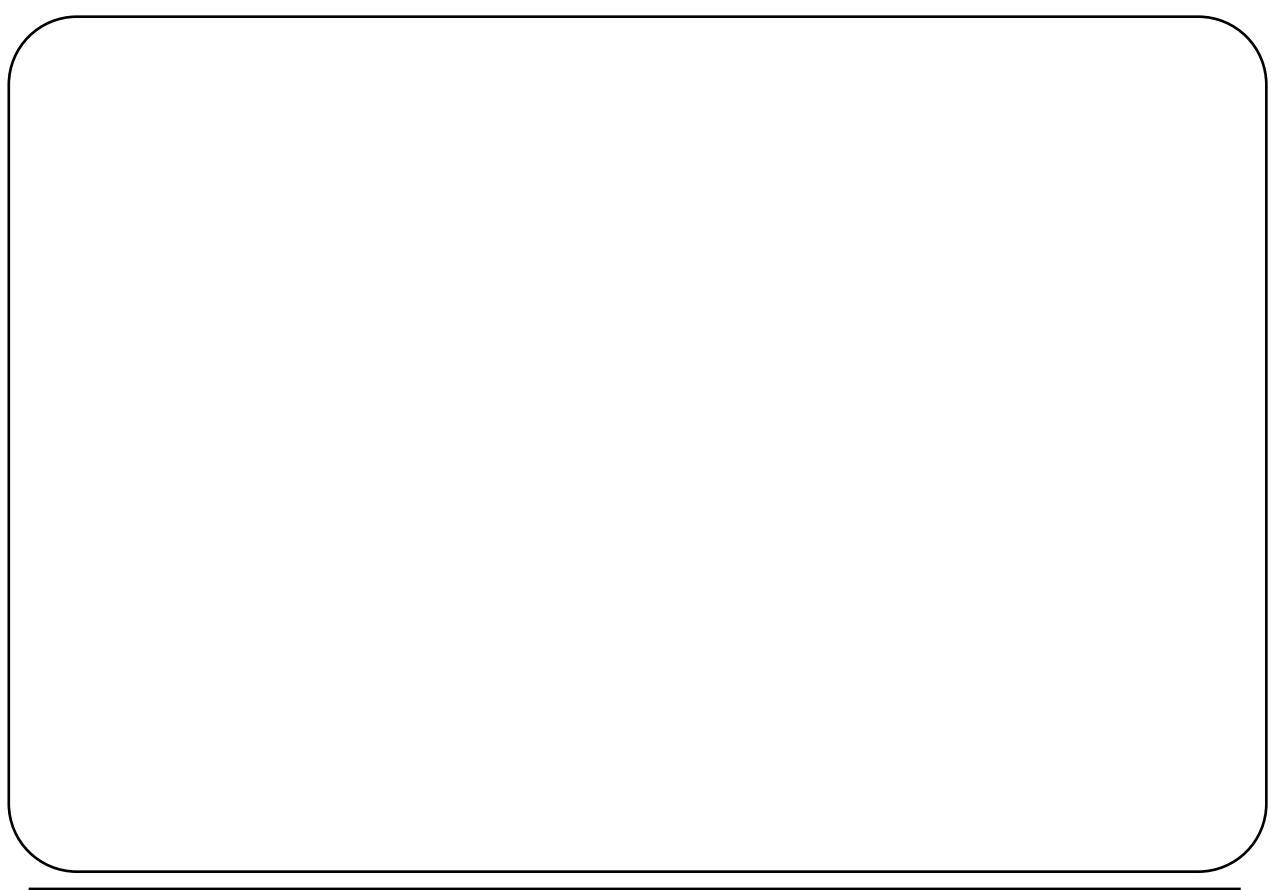
```
class Node
```

in the List class. This defines Node as a nested class.

The Node class is defined using double class scope resolution:

```
class List::Node
{
    // attributes and methods
}
```





# 13

## Aggregation

- **Classes within Classes**
- **Member Functions**
- **Constructor Issues**
- **Destructor Issues**



13

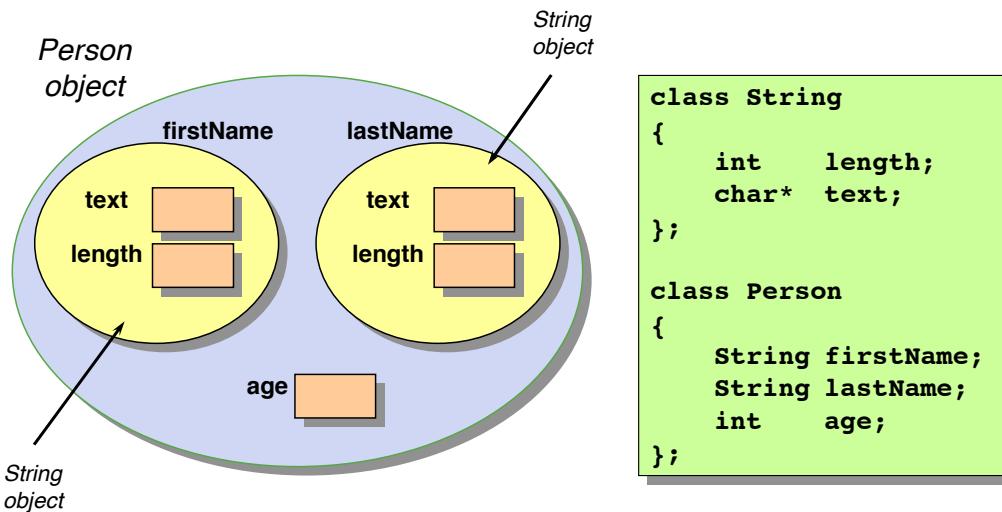
Much of the power of C++ lies in the ability to work at higher and higher levels of abstraction as we define more and more useful classes. Object Orientation provides two mechanisms for doing this: Aggregation and Inheritance. Inheritance will be discussed later in the course.

Aggregation (sometimes referred to as Composition) uses the well tried technique of creating new classes by combining old ones. Aggregation differs from inheritance in that the new class is built from scratch and not by adding attributes and methods to an existing class.

We use aggregation when an object is built from many sub-objects as in the case of a car object built from several wheel objects, a body object, an engine object, etc. Each of these sub-objects may themselves be aggregates: an engine object consists of a block object, piston objects, etc.

We use aggregation when an object HAS A number of component objects. We use inheritance when an object IS A kind of other object.

## Classes within Classes



Aggregation is the process of defining a new class that incorporates older classes. The new class and the older classes are not related in any way (unlike in inheritance).

Here is a simple example. The **Person** class needs to use two strings for its data members. Rather than using a `char*` data type and reinventing the code to put a character array on the heap, it is more sensible to utilize the **String** class developed earlier.

Utilizing the **String** class enables us to work at a higher level of abstraction, saves rewriting code and reduces time spent in testing.

The memory layout of the **Person** object is shown above. Note each **String** object is effectively a sub-object of the **Person** object.

## Member Functions

### *Person.hpp*

```
class Person
{
private:
    String firstName;
    String lastName;
    int age;
public:
    Person(String, String, int);
    void Display() const;
};

void Person::Display() const
{
    cout << "Person";
    firstName.Print();
    lastName.Print();
    cout << "Age" << age << endl;
}
```

### *String.hpp*

```
class String
{
private:
    int length;
    char* text;
public:
    void Print() const;
};
```

Copyright ©1994-2011 CRS Enterprises

173

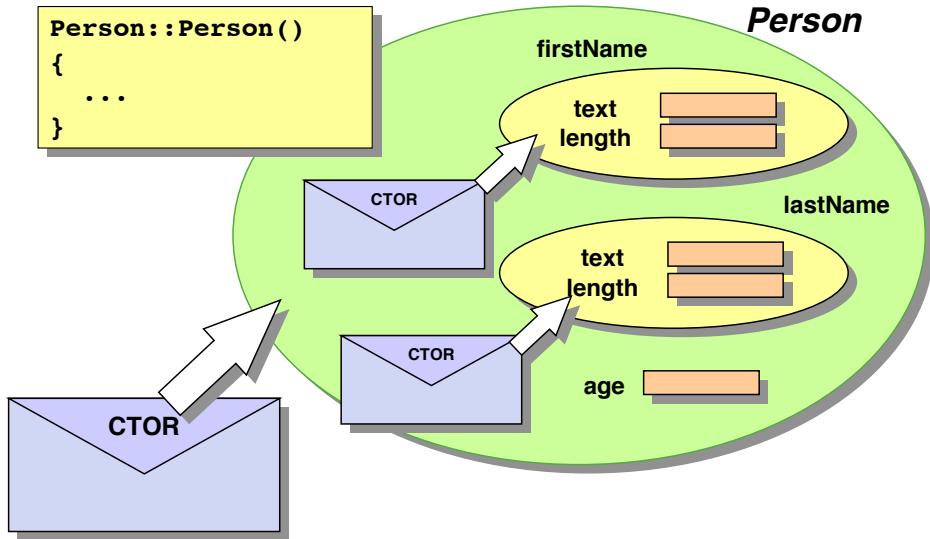
We begin to see the advantages of aggregation when we realize that we can send messages to sub-objects.

Here, we are writing a `Display` member function for the `Person` class. The `age` component is a built in type and therefore we must write code to print the object's age. However, `firstName` and `lastName` are user defined types. This means we can send these sub-objects the message:

`firstName.Print()`  
`lastName.Print()`

without having to write code to extract the character array from the heap (this is indeed fortuitous since the `String` class declares its internal data members as private and hence we would not have been able to access its data directly).

## Constructor Issues ...



Copyright ©1994-2011 CRS Enterprises

174

The initialization of the Person object proceeds by sending a constructor message to the Person object. The age attribute is initialized directly, but both the internal string objects have their data members (text, length) encapsulated and hence cannot be initialized directly. Instead these string must themselves be sent a constructor message. Thus the original constructor splits into other constructors to initialize sub objects. This procedure continues until only simple attributes remain to be initialized.

## ... Constructors Issues

```

class Person
{
private:
    String firstName;
    String lastName;
    int age;
public:
    Person(String, String, int);
    void Print();
};

Person::Person(String s1, String s2, int a)
:
    firstName (s1),
    lastName (s2),
    age (a)
{ }

```

Copyright ©1994-2011 CRS Enterprises

175

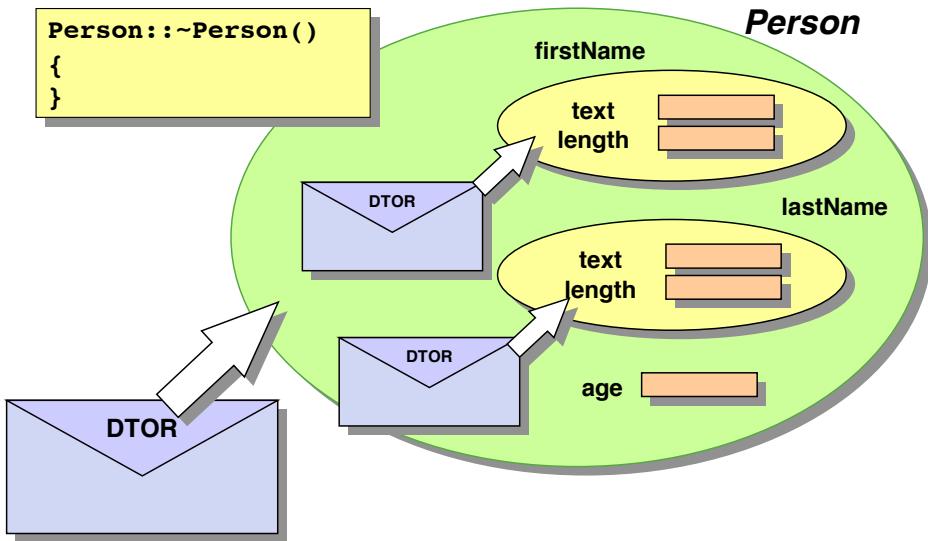
When we construct Person objects we are only responsible for constructing the components that are not sub-objects (viz. the age component). Construction of the name component can be delegated to the String constructor.

The part of the initialization list in the Person constructor:

**name(theAge),**

calls the String constructor directly. We do not need (or want) to initialize the String component directly. The details of the String sub-object are hidden from us; it is not our concern.

## Destructor Issues



Copyright ©1994-2011 CRS Enterprises

176

Returning to our Employee and String example, note that the Employee class did not provide a destructor. This is eminently reasonable because the Employee class does not use the heap. Or does it?

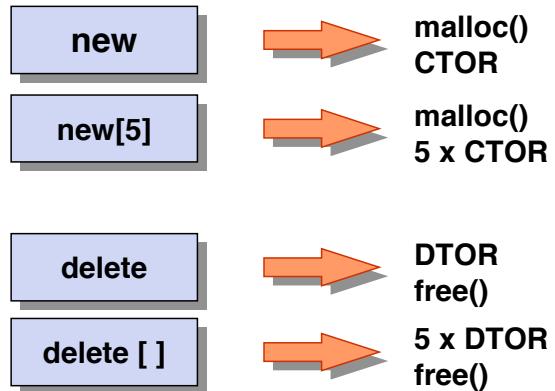
In fact the String sub-object does use the heap, so perhaps we do need a destructor after all.

Fortunately, we don't have to worry. While it is true that one of our sub-objects uses the heap, at our level of abstraction we should not be aware of how the underlying objects operate. Nevertheless, the compiler is aware of what is going on and therefore automatically generates a do nothing destructor whose only purpose is to call the String destructor which does the real work.

In the general case, destructors are automatically called for all sub-objects. The class designer need only provide an explicit destructor if the class uses the heap directly.

## Using the Heap

- **new and delete**



Copyright ©1994-2011 CRS Enterprises

177

The new and delete operators are used to create and destroy objects on the heap. The new[ ] and delete[ ] operators are used to create and destroy arrays of objects on the heap.

The new operator calls malloc to allocate the storage for the new object and then implicitly calls the appropriate constructor to initialize the object. Likewise delete implicitly calls the destructor to free up secondary resources before calling free to deallocate the objects memory.

With arrays of objects the process is similar; malloc and free are used to allocate/deallocate heap memory, but since an array consists of several objects new[ ] and delete[ ] call a constructor/destructor for each object. If you allocate an array with new[ ], but deallocate with delete instead of delete[ ], only one destructor is called and this may result in memory leakage.

## Deleting Arrays from the Heap

```

int main()
{
    Person* ptr;

    // allocate array of objects on heap
    ptr = new Person[3];

    ptr[0] = Person("John", "Black", 50);
    ptr[1] = Person("Susan", "Green", 25);
    ptr[2] = Person("Chris", "White", 41);

    // code to use array of objects

    // deallocate array of objects from heap
    delete [ ] ptr;
}

```

Copyright ©1994-2011 CRS Enterprises

178

Consider creating three Person objects as part of an array allocated on the heap. Each Person object will be initialized by its default constructor:

**Person::Person()**

We can reinitialize each object by calling a three parameter constructor:

**Person:: Person(const char\*, const char\*, int)**

The array is now ready to be used in our code (not shown). When we come to delete the array we must call a destructor for each element of the array. The code

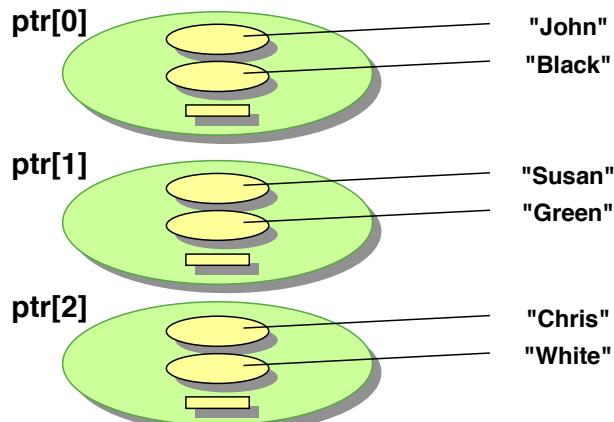
**delete [ ] ptr**

sends a destructor message to each element of the array. There is no need to specify the number of elements to delete; the compiler remembers the size of the array.

## Does your Heap Leak?

- Spot the memory leak

```
Person* ptr = new Person[3];
...
delete ptr; // [ ] missing
```



Copyright ©1994-2011 CRS Enterprises

179

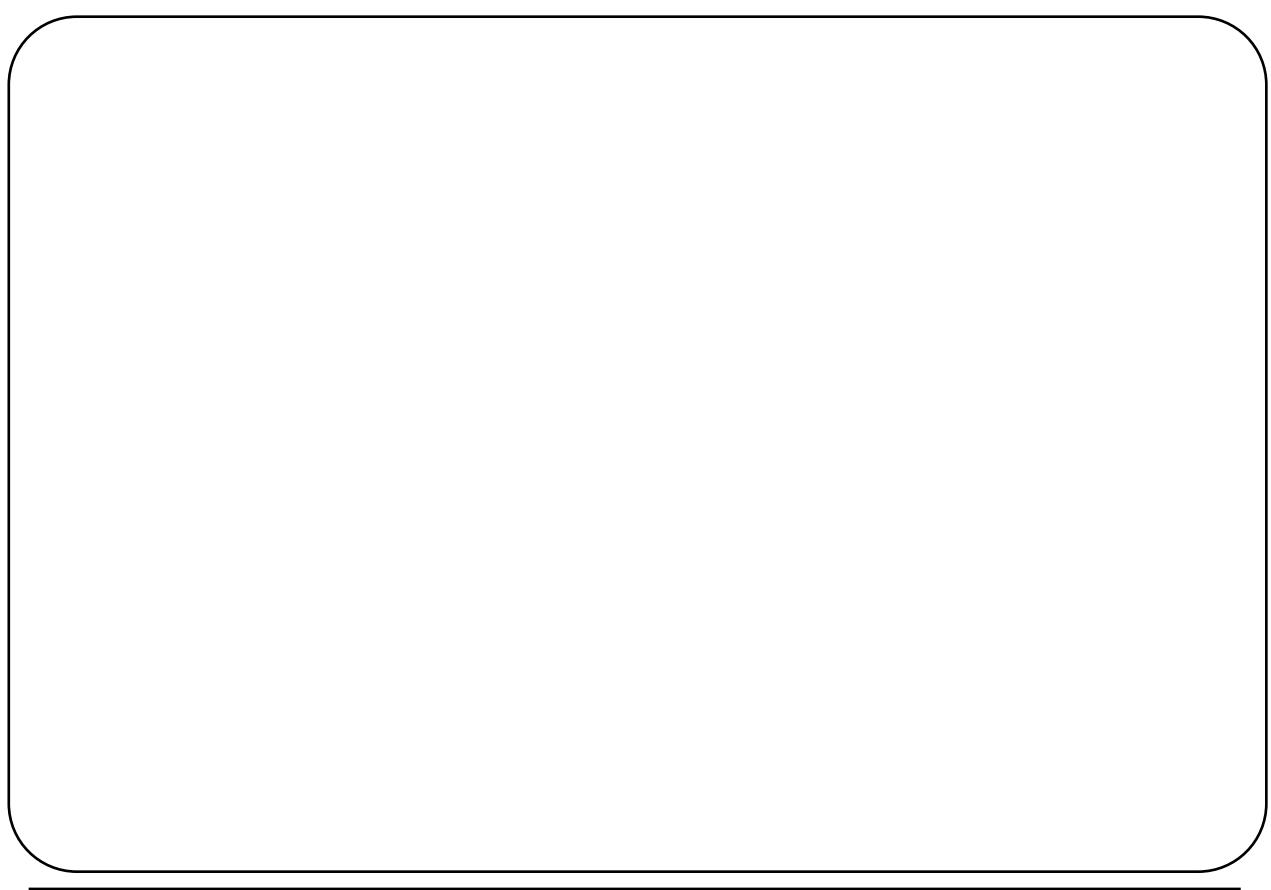
It was important to call

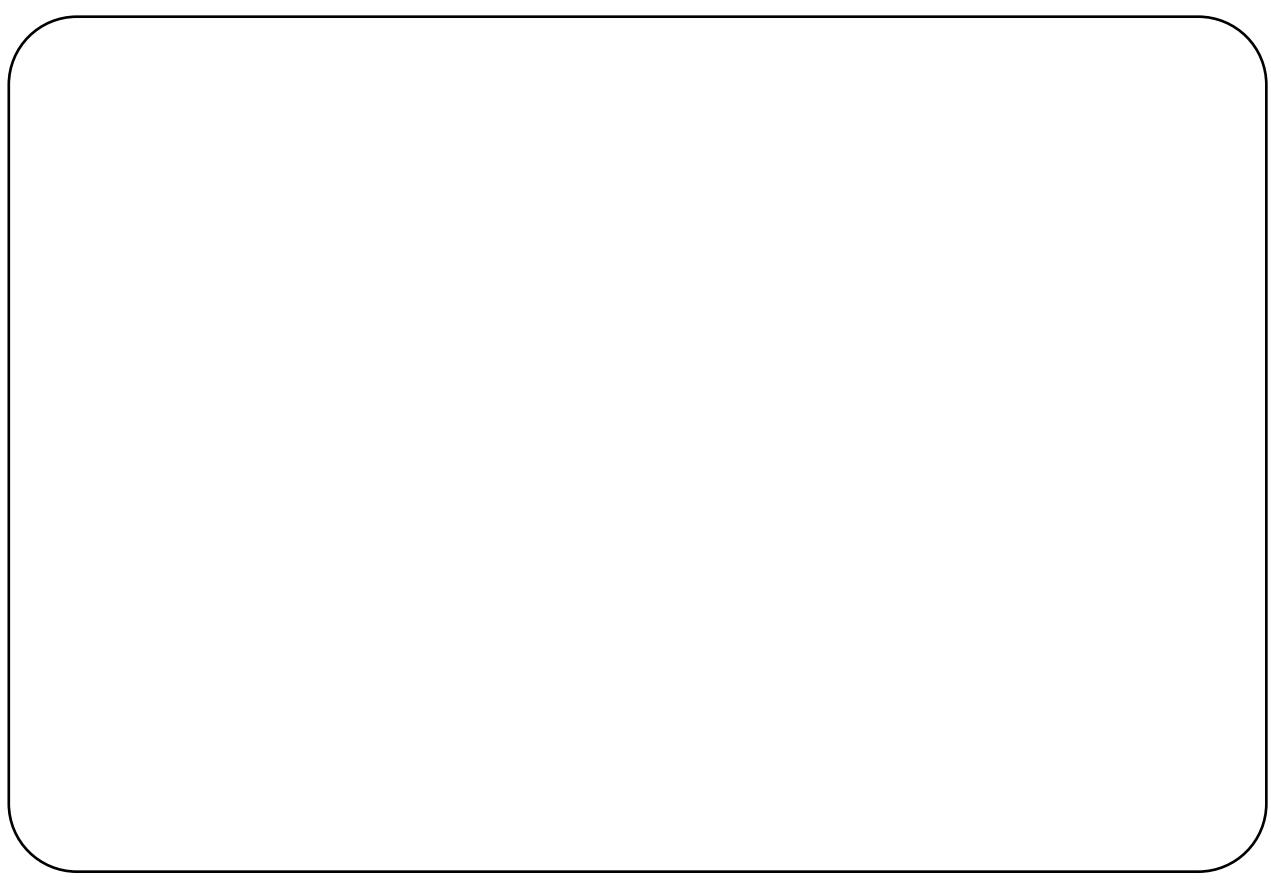
**`delete[] ptr`**

in the previous example to ensure the heap was cleaned up properly. Remember that `delete[]` will remove the 3 `Employee` objects from the heap, but will not automatically remove the 6 character arrays.

To remove these 6 character arrays from the heap we must make use of our destructors. The `delete[]` operator calls 3 `Employee` destructors in addition to cleaning up the array of `Employee` objects. Each `Employee` destructor is empty, but calls 2 `String` destructors. It is the `String` destructors that clean up the 6 character strings that were placed on the heap (that were created by the `String` constructors).

Note the importance of the `[]` in the `delete` operator. Without the brackets all `Employee` objects will still be removed from the heap, but only one `Employee` destructor is called and therefore only the first 2 character arrays ("John" and "Black") get cleaned up. The remaining character arrays are left on the heap!





# 14

## More on References

- **References to Objects**
- **Reference Returns**



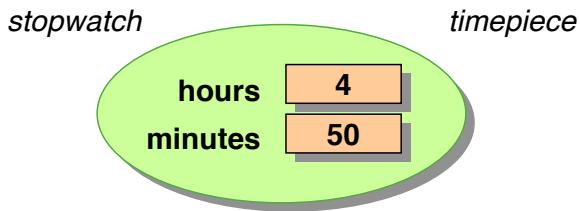
# 14

In this chapter we investigate C++ references. A reference is a simple data type that is less powerful but safer than the pointer type inherited from C. The name C++ reference may cause confusion, as in computer science a reference is a general concept data type, with pointers and C++ references being specific reference data type implementations.

References are used in three areas:

- **Providing alternative names (aliases) for existing variables**
- **Supporting function calls by reference**
- **Allowing functions to return references**

## References to Objects



```
Time stopwatch(4,50);
Time& timepiece = stopwatch;

stopwatch.SetToZero();
timepiece.SetToZero();

stopwatch.DisplayTime();
timepiece.DisplayTime();
```

Copyright ©1994-2011 CRS Enterprises

184

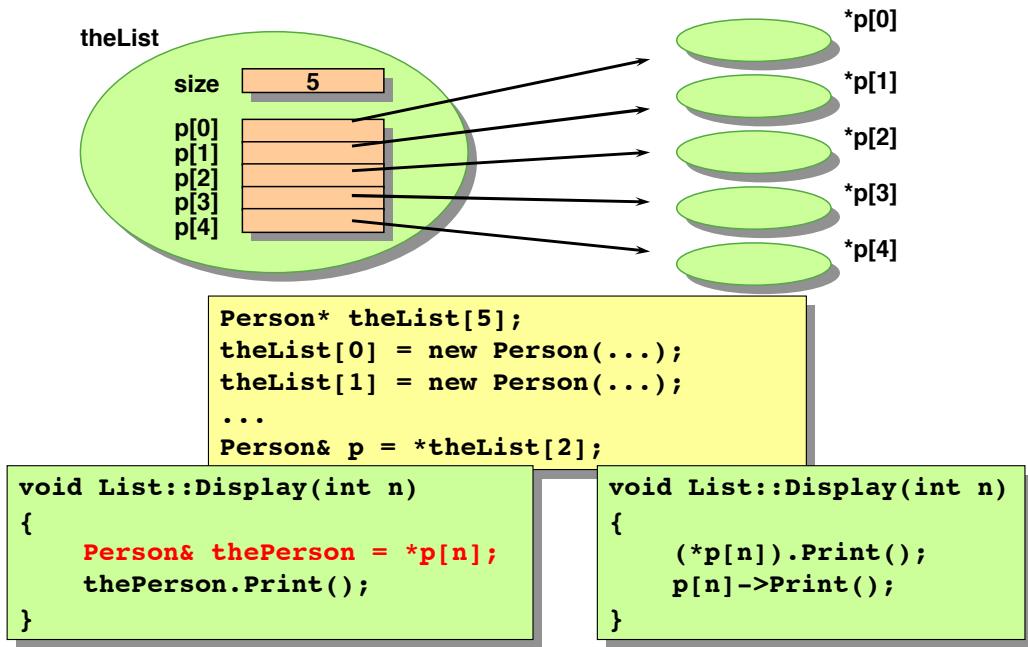
When defining a reference to a user defined object the situation is the same as for a built in type. The reference is merely an alias for the original object.

However, it is very important to realize that the compiler does not create a new object when we declare a reference. Therefore there is nothing to be initialized and consequently no constructor is called.

Not only is the constructor not called when the reference is declared, but nor is the destructor called when the reference goes out of scope.

Although the reference name and the original name refer to the same object, the two names may have different scope. We will investigate this possibility later in this chapter. Note that a constructor is called when the original name is defined and a destructor is called when the original name goes out of scope.

## Simplifying Notation



Copyright ©1994-2011 CRS Enterprises

185

Sometimes it is useful to use a reference just to simplify coding. In the example opposite we have an object theList that maintains an array of 5 pointers to dynamic objects. Each object has the usual alias involving the \* symbol but in this case things are made more difficult because we are using an array.

The message

**theList.Display(2)**

invokes the Display method with n equal to 2. Inside the method we send a message to the object pointed at by p[2]. This object is known as

**\*p[2]**

We can send a Print message to p[n] using the notation

**(\*p[n]).Print()**

or

**p[n]->Print();**

but in both cases this involves complicated notation. A better solution is to rename the object using a reference

**Person& thePerson = \*p[n];**

From now on we can refer to \*p[n] simply as thePerson. When we send a Print message to thePerson we simply write

**thePerson.Print()**

## Reference Returns ...

```

class Array
{
private:
    int a[5];
public:
    Array();
    void Print();
    int& Select(int);
};

Array::Array()
{
    for (int i = 0; i < MAX; i++)
        a[i] = 5;
}

void Array::Print()
{
    for (int i = 0; i < MAX; i++)
        cout << '\t' << a[i];

    cout << endl;
}

```

Functions can also return references in C++. This is illustrated in the next 2 slides.

This slide shows a class `Array` that has a constructor and two methods. The `Print` method is straightforward, but the `Select` function is of interest since it returns a reference to an integer. The code for this method is shown overleaf. The class has attributes that wrap a C style array of 5 integers.

## ... Reference Returns

```
int& Array::Select(int index)
{
    if (index >= 0 && index < 5)
        return a[index];
    else
        throw "Out of bounds";
}
```

```
int main()
{
    Array matrix;
    matrix.Print();
    matrix.Select(1) = 10;
    matrix.Print();
    matrix.Select(9) = 10;
    matrix.Print();
}
```

Copyright ©1994-2011 CRS Enterprises

187

Here is the code for the Select method. Select normally returns a reference to one of the integers in the array a. Select also performs bounds checking on the array. If the index selected is out of range and throws an exception in that case,

In the main function, the matrix object is sent the Select message twice. On the first occasion the index is in range and a reference to a[1] is returned. Thus the statement,

**matrix.Select(1) = 10;**

is equivalent to:

**a[1] = 10;**

since the return is an alias for a[1].

On the second occasion the index is out of bounds, so an exception is thrown (see later in the course).

## ... Reference Returns

- If we rename the method from Select to operator[ ] ...
  - we overload the [ ] operator

```
int& Array::operator[](int index)
{
    if (index >= 0 && index < 5)
        return a[index];
    else
        throw "Out of bounds";
}
```

```
int main()
{
    Array matrix;
    matrix.Print();
    matrix[1] = 10;
    matrix.Print();
    matrix[9] = 10;
    matrix.Print();
}
```

Copyright ©1994-2011 CRS Enterprises

188

The Select method is equivalent to overloading the [ ] operator. Indeed if we rename Select to operator [ ], the compiler lets us write code using the more natural syntax shown above:

**matrix[1] = 10;**

## Returning Local References

- Beware returning a reference to a local object

```

class Date
{
private:
    int day;
    int month;
    int year;
public:
    Date();
};

Date& Function ()
{
    Date result;
    // body of function
    return result;
}

```

```

int main()
{
    Date s;
    s = Function ();
}

```

Copyright ©1994-2011 CRS Enterprises

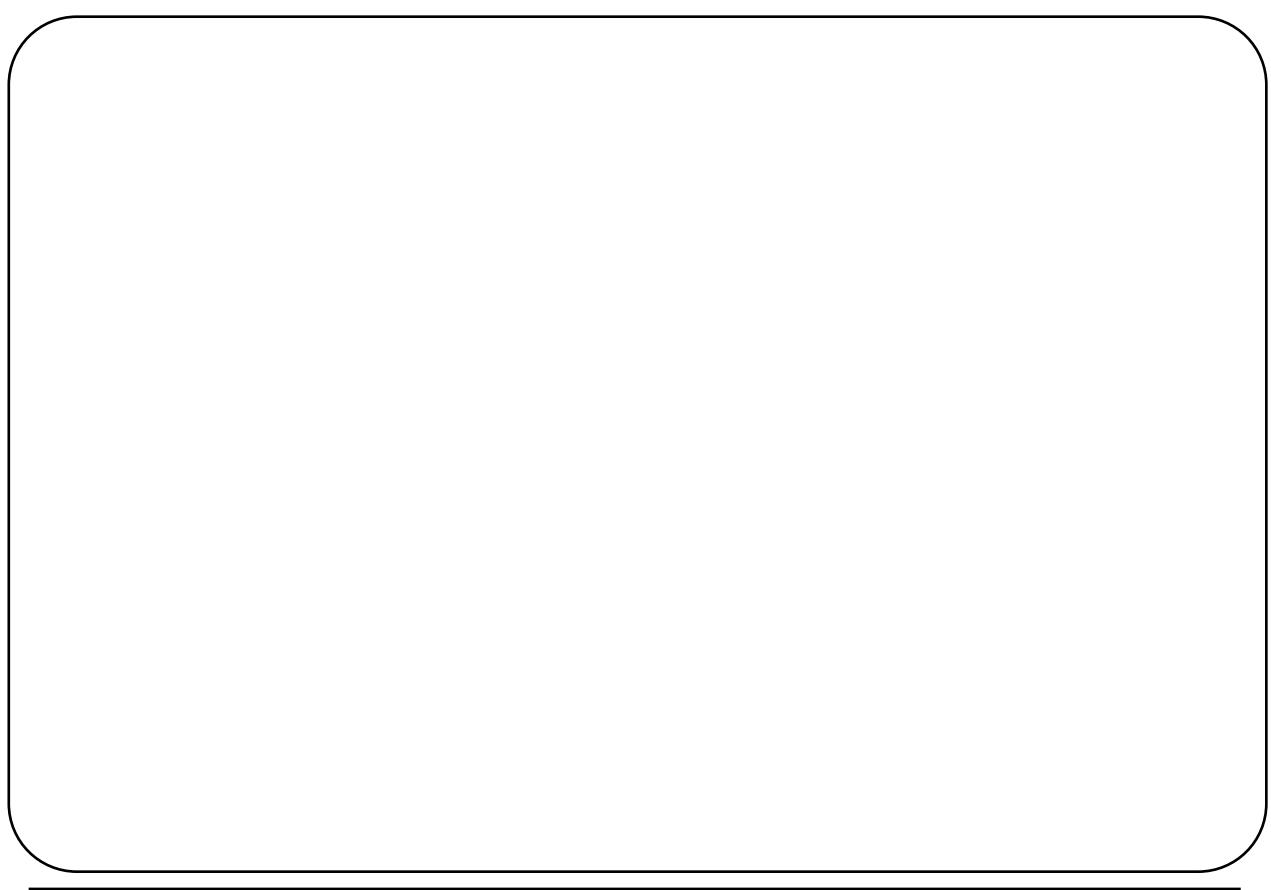
189

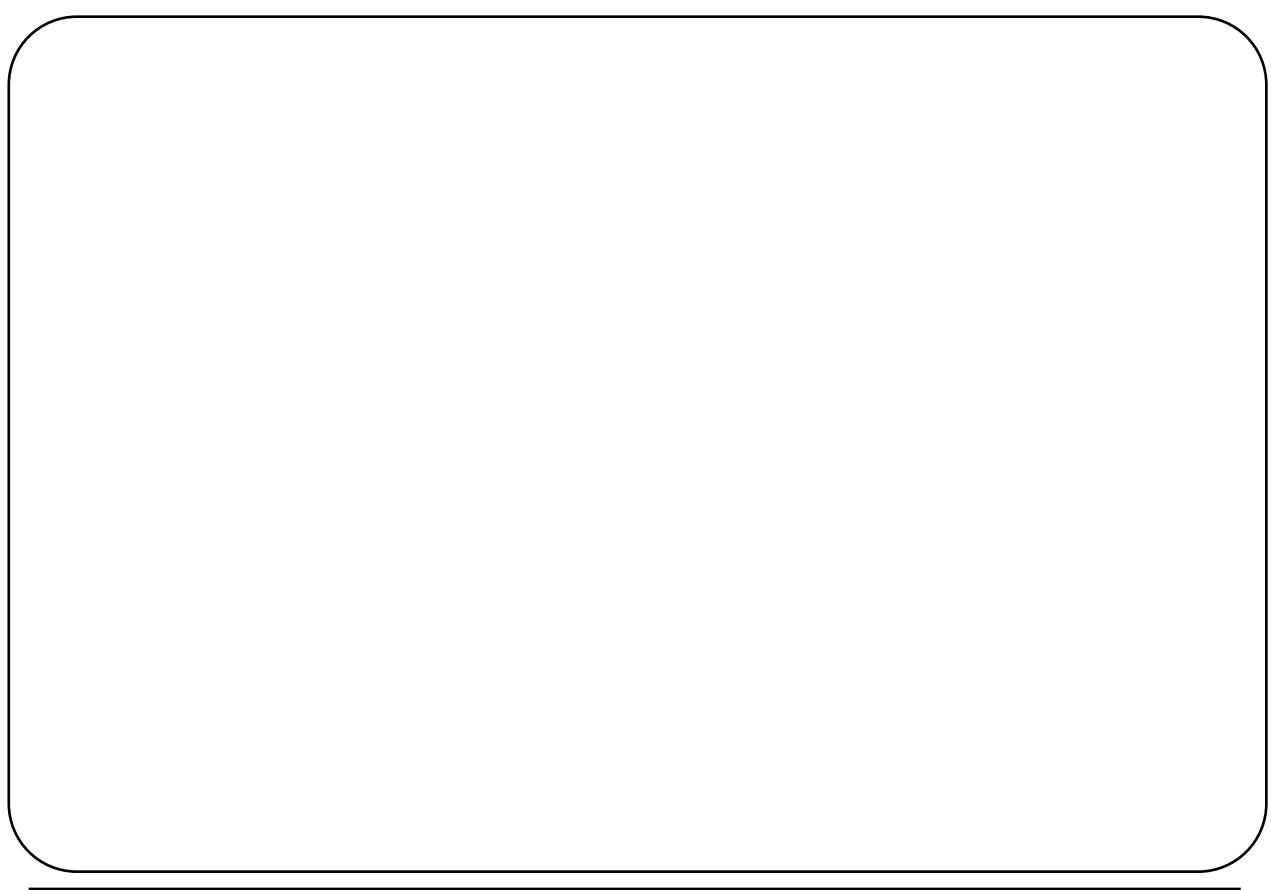
As we have seen, care must be exercised when passing objects to functions by value and returning objects by value. This example shows that we must be equally vigilant with references.

Consider the example above. The result object is constructed as a local object in the function. When we execute:

**return result;**

we make the anonymous return area a reference to the result to avoid making a copy of the object and hence avoid invoking its constructor. Unfortunately, when the function completes result is destroyed. The anonymous return area is left referencing a non existent object!





# 15

## Operator Overloading

- **Binary Operators**
- **Unary Operators**
- **Overloading Member Functions**
- **Overloading Friend Functions**



# 15

Operator Overloading in C++ permits the programmer to redefine the meaning of operators on a class by class basis. This allows user defined classes to work with operators in the same way as built in data types.

Redefining the meaning of operators is achieved by providing methods for a class. A new keyword operator is used to give these methods special names. For example,

**operator+()**

is the name of the method that redefines the + operator for a class.

## Operator Overloading ...

- C++ allows us to redefine operators
  - the compiler employs special conventions
- Usually you can overload using ...
  - a free function (recommended) or
  - a member function

```
Time t, t1, t2;

t = t1 + t2;

// translates to either
t = operator+(t1, t2);
// or
t = t1.operator+(t2)
```

The diagram illustrates the translation of the expression `t = t1 + t2;`. It shows two equivalent forms: `t = operator+(t1, t2);` and `t = t1.operator+(t2);`. Arrows point from the original expression to each of these forms, with the text "free function" next to the first and "member function" next to the second.

C++ allows us to redefine operators using special compiler conventions. With most operators you can overload using either a free function or a member function, but not both.

It is generally recommended to use free functions in preference to using member functions mainly because it is easier to make free function generic (i.e. using templates). There are other advantages as well and these will be discussed later in the chapter.

## Using Free Functions

```

Time.hpp
class Time
{
private:
    int hours;
    int minutes;
public:
    Time(int, int);
    Time();
};

Main.cpp
int main()
{
    Time t1(3,40);
    Time t2(5,15);
    Time t;

    t = operator+(t1, t2);
}

```

The diagram illustrates the use of a free function for operator overloading. On the left, **Time.hpp** defines a **Time** class with private members **hours** and **minutes**, and public constructors **Time(int, int)** and **Time()**. On the right, **Main.cpp** shows the **main** function where a new **Time** object **t** is created by assigning the result of the **operator+(t1, t2)** call to it.

Consider how we might write a function to perform the add operation for a **Time** class if operator overloading was not supported by C++.

Here we have three **Time** objects and we can perform an addition by:

**t = Add (t1, t2);**

By convention we rename the function as follows:

**t = operator+(t1, t2);**

and this allows us to redefine the **+** operator for the **Time** objects.

Notice the use of references for efficiency (this saves the overhead of creating a copy of **t1** and **t2** on the stack). The **operator+** function also returns **Time** object. This object is returned by value and not by reference because it is a new object (it is a copy of some other **Time** object). The assignment operator (not overloaded here!) copies this object to **t**.

## Implementing operator+

*Time.cpp*

```
Time operator+(const Time& x1, const Time& x2)
{
    Time result;

    result.hours = x1.hours + x2.hours;
    result.minutes = x1.minutes + x2.minutes;

    if (result.minutes >= 60)
    {
        result.minutes -= 60;
        result.hours++;
    }

    return result;
}
```

*doesn't  
compile !*

Copyright ©1994-2011 CRS Enterprises

196

The implementation of the operator+ method is shown above.

We first create a temporary object result which will be used to hold the result of the + operation. Because result is declared as a local object we must remember that it will be destroyed at the end of the function.

Just before the function ends we create a copy of result with the statement:

**return result;**

This happens because we are returning result by value. Note that the Time constructor is called to build the copy of result. It is this copy that is assigned to t when we return to the calling program:

**t = operator+(t1, t2);**

Note that we still have a big problem - we are accessing private data from a global function. The code doesn't compile!

## Introducing Friends

- **By making the free function a friend ...**
  - it has the same access rights as a normal method
  - and can now access private data
- **Is this really above board ...**
  - a hack?
  - or syntax sugar?

```
class Time
{
    friend Time operator+(const Time& x1, const Time& x2);
private:
    int hours;
    int minutes;
public:
    Time(int, int);
    Time();
};
```

Copyright ©1994-2011 CRS Enterprises

197

We can get the preceding code to compile by making our free function a friend of the Time class. Friend functions are granted the same access as regular member functions and hence our code will now compile.

Questions still remain on whether this is reasonable. Is it a hack or merely syntax sugar? Programmers have been arguing about this since it was first introduced into C++, but nowadays it is generally accepted that this is OK.

The operator+ free function can only be used with Time objects and it is really part of the class definition. It is only the language syntax that prevents us stating this directly. Friend functions were introduced to address this inconsistency.

If it really upsets you, then add getter and setter methods to the class!

## Using Methods ...

- Alternatively implement using member functions

<i>Time.hpp</i>	<i>Main.cpp</i>
<pre>class Time { private:     int hours;     int minutes; public:     Time operator+(const Time&amp; x); };</pre>	<pre>int main() {     Time t1(3,40);     Time t2(5,15);     Time t;     t = t1 + t2;     // t = t1.operator+(t2); }</pre>

Alternatively we can implement the + operator using a member function. The code is equivalent to the friend function shown earlier, but there are some minor syntactical differences.

The statement:

**t = t1 + t2;**

is completely equivalent to:

**t = t1.operator+(t2);**

## ... Using Methods

### *Time.cpp*

```
Time Time::operator+(const Time& x)
{
    Time result;

    result.hours = hours + x.hours;
    result.minutes = minutes + x.minutes;

    if (result.minutes >= 60)
    {
        result.minutes -= 60;
        result.hours++;
    }

    return result;
}
```

The implementation of the operator+ as a method is shown above.

Note that it is nearly identical with the free function implementation. The main difference is that *\*this* replaces the first parameter to the free function and we don't need to declare the method as a friend.

## Comparing Techniques ...

### Methods

```
t = t1 + t2;
// t = t1.operator+(t2)
```

```
class Time
{
public:
    Time operator+ (const Time&);

    Time Time::operator+(Time& x)
    {
        // this points at object
        // receiving message
    }
}
```

### Free Functions

```
t = t1 + t2;
// t = operator+(t1, t2)
```

```
class Time
{
public:
    friend Time operator+
        (const Time&, const Time&);

    Time operator+
        (const Time& x1, const Time& x2)
    {
        // no this parameter
    }
}
```

The previous examples showed how to overload the binary + operator as a method or a free function. Most binary operators can be overloaded in this manner.

Free functions are overloaded functions that are distinguished by their signature (parameter types). Often there will be a multitude of friend functions for each operator, each with a different signature.

Notice that since free functions are not associated with a class, there is no *\*this* object to send a message to. Both objects involved in the addition must be passed as parameters to the free function.

## ... Comparing Techniques

### Methods

```
Time Time::operator+(const Time& x)
{
    // this points at object receiving message
    Time result;
    result.hours = hours + x.hours;
    result.minutes = minutes + x.minutes;
    return result;
}
```

t1 + t2

### Friend Functions

```
Time operator+(const Time& x1, const Time& x2)
{
    // no this parameter
    Time result;
    result.hours = x1.hours + x2.hours;
    result.minutes = x1.minutes + x2.minutes;
    return result;
}
```

t1 + t2

Copyright ©1994-2011 CRS Enterprises

201

When we implement operator overloaded member functions we must use the scope resolution operator to tie the method to the class:

**Time::operator+ (const Time& t)**

We implement friend functions in a similar manner to methods. The important difference is that there is no \*this object receiving a message and therefore both objects are passed as parameters. Furthermore, the friend function is global and must not be tied to a class:

**operator+ (const Time& t1, const Time& t2)**

Nevertheless, the data types of parameters to the friend function ensure that the function only works with the given class.

Notice that the code in the body of the friend function is now symmetric:

**result.hours = x1.hours + x2.hours;**  
**result.minutes = x1.minutes + x2.minutes;**

# Unary Functions

## Methods

```
class Time {
public:
    Time operator-();
};

Time Time::operator-()
{
    // this points at object receiving message
}
```

t1 = -t2;  
// t1 = t2.operator-()

## Friend Functions

```
class Time {
public:
    friend Time operator-(const Time&);

};

Time operator-(const Time& x)
{
    // no this parameter
}
```

t1 = -t2;  
// t1 = operator-(t2)

Unary operators are treated in a similar way to binary operators, but this time there is only one operand to consider.

For overloaded operator methods the message is sent to the one and only operand and hence there are no explicit parameters in the call.

With friend functions there is no message to send (because friend functions are not methods) and therefore the operand is passed as an explicit parameter.

# Implementation

## Methods

```
Time Time::operator-()
{
    // this points at object receiving message
    Time result;
    result.hours = -hours;
    result.minutes= -minutes;
    return result;
}
```

-t2;

## Free Functions

```
Time operator-(const Time& x)
{
    // no this parameter
    Time result;
    result.hours = -x.hours;
    result.minutes= -x.minutes;
    return result;
}
```

-t2;

The function prototypes for the unary operators are very similar to those used for binary operators. The only difference is that there is one less parameter in both cases.

The method passes one parameter implicitly, the friend function passes one parameter explicitly.

## Restrictions

- **Can't overload built-in types**
- **Can't create new operators**
- **Can't change operator precedence**
- **Can't overload some operators**
- **Unary operators stay unary**
- **Binary operators stay binary**

Although operator overloading is a powerful feature in C++, certain restrictions had to be enforced:

The built in types must work as they did in C. It is not possible to change the meanings of operators for built in types.

Only the existing operators can be overloaded. You are not permitted to invent new operators.

The precedence table for operators remains in force even if you change the meaning of an operator for a given class.

Some operators are exempt from being overloaded. For example, the scope resolution operator :: and the dot operator . cannot be overloaded; these operators are used to redefine the meaning of the operators you can overload and chaos would ensue if these operators could be redefined! The complete list of exempt operators is given below:

- **member selection**
- **:: scope resolution**
- **? : conditional expression**
- **# convert to string (preprocessor)**
- **## token pasting (preprocessor)**

Finally, if a unary operator is being redefined then the redefined operator must also be unary; similarly with binary operators.

Despite these restrictions, operator overloading is both powerful and flexible. User defined types are effectively treated on a par with built in types.

## Side Effect Operators

```
Date d1, d2;
d1 = ++ d2;
```

*calls*  
`Date::operator++()`

```
Date d1, d2;
d1 = d2++;
```

*calls*  
`Date::operator++(int)`

```
Date& Date::operator++()
{
    // increment *this
    // return *this
}
```

```
Date Date::operator++(int)
{
    // take copy of *this
    // increment *this
    // return copy
}
```

The prefix and postfix side effect operators can be overloaded separately in C++ by pretending that the postfix operator carries an unused integer parameter. This makes the signature of the postfix operator different to the prefix operator. This is hardly an elegant solution, but was regarded as simpler than inventing a special syntax for overloading side effect operators.

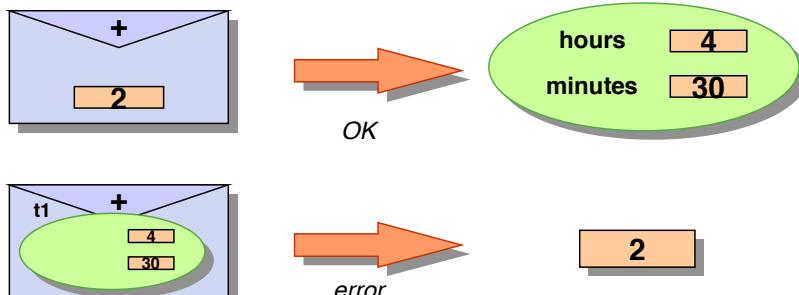
Note the complications with the postfix implementation. The return value from the function needs to be the `*this` object before it is incremented. The only way to arrange this is to create a copy of `*this` and then return this copy. Since the copy is a local object it cannot be returned by reference. Clearly the prefix implementation is much simpler.

## Problems with Methods

*Main.cpp*

```
Time t;
Time t1(4,30);

t = t1 + 2;      // t = t1.operator+(2)
t = 2 + t1;      // t = 2.operator+(t1)
```



There is an important caveat with operator overloading arising from the fact that methods are asymmetric. Consider the example shown opposite. If a Time object is added to an integer, the `operator+()` message can be sent to the Time object. However, if the arguments are reversed we run into problems. Now we need to send the message to an integer; this is not possible since an integer is a built in type (and has no methods).

Hence methods can only be used when the first argument is a class object. If the first argument is a built in type then a friend function must be used.

This situation will arise quite often in practice and therefore some people prefer always to use friend functions when overloading operators for a class. It should be noted that friend functions are always public and cannot be inherited by derived classes; so use methods if you have a choice.

## Method or Friend?

### Methods

Date d, d1, d2;  
Time t, t1, t2;

d1 + d2	Date:: operator+(Date&)
d + t	Date:: operator+(Time&)
d + 20	Date:: operator+(int)
t + d	Time::operator+(Date&)
20 + d	N/A

### Free Functions

d1 + d2	operator+(Date&, Date&)
d + t	operator+(Date&, Time&)
d + 20	operator+(Date&, int)
t + d	operator+(Time&, Date&)
20 + d	operator+(int, Date&)

When implementing operator overloading we often have a choice between using a method or using a friend function. For example the 3 expressions shown opposite:

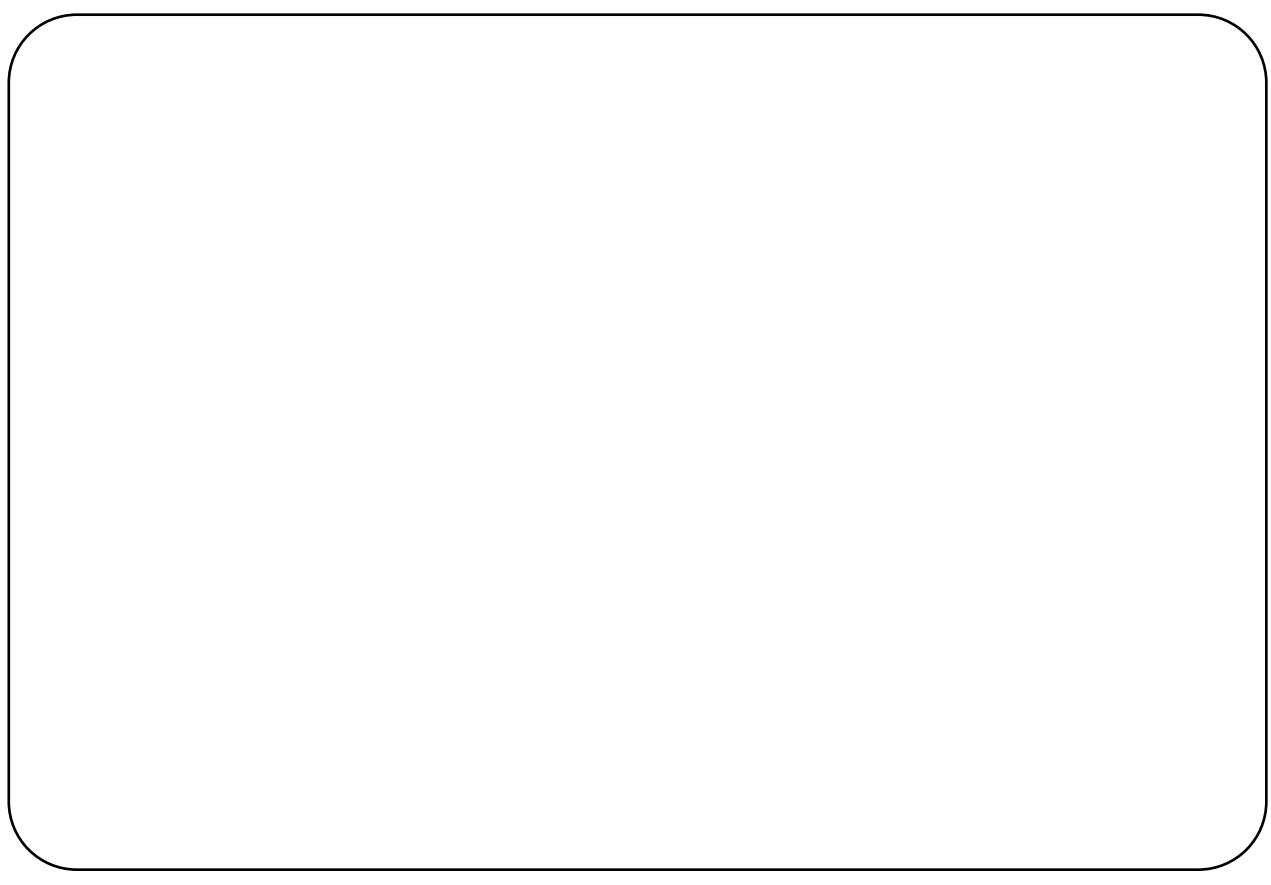
**d1 + d2**  
**d + t**  
**d + 20**

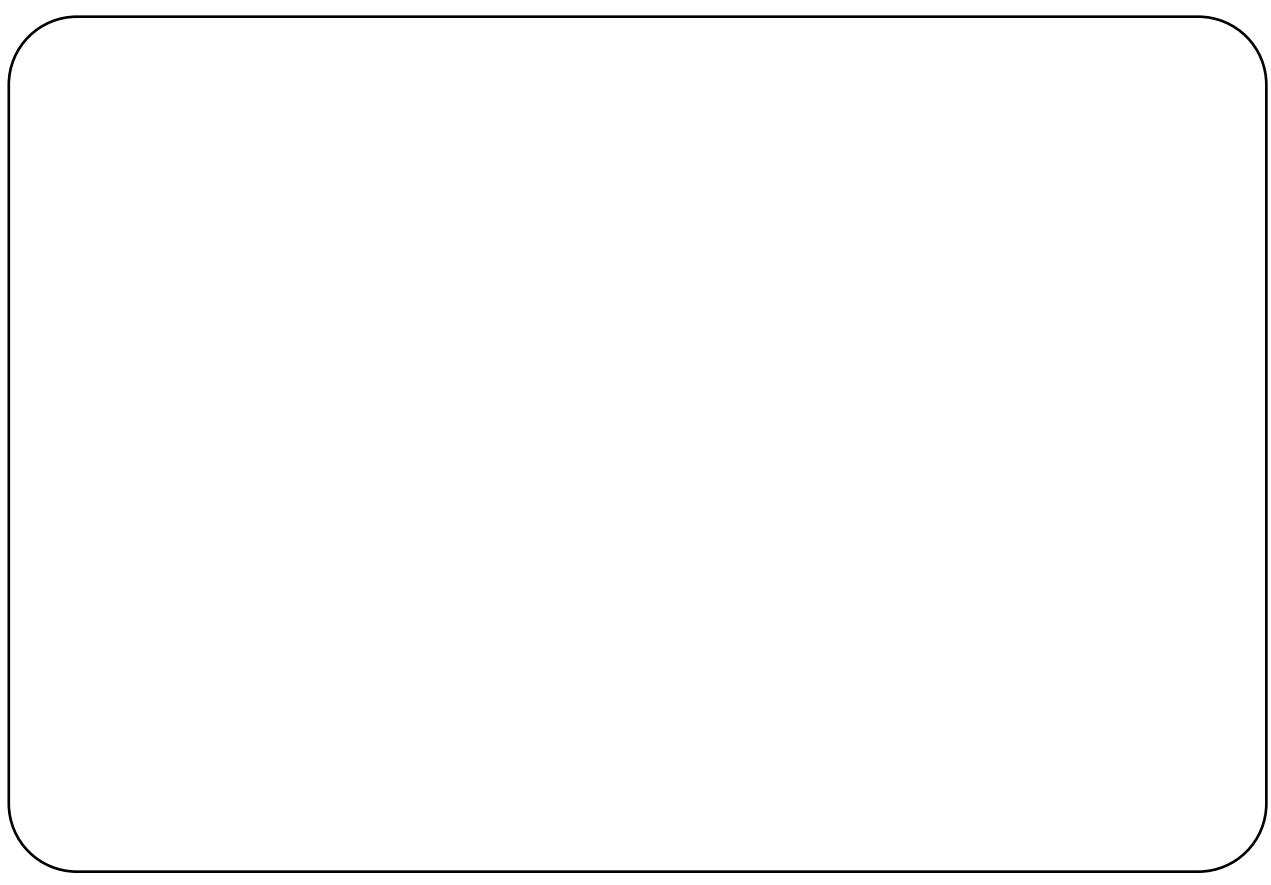
can be implemented as either. The compiler has no preference between methods and friends, but will not permit both implementations. The choice ultimately depends on personal preference.

There are examples where you must use a friend function because the method is undefined as in:

**t + d**  
**20 + d**

Remember that friend functions violate encapsulation by accessing data directly. Friend functions are global functions with special access to the class. Sometimes it is possible to implement operator overloading in terms of global functions that are not friends of a class. This implies that the function does not violate encapsulation. We can achieve this by arranging for the function to call a member function internally.





# 16

## Shallow and Deep Classes

- **Problems with Shallow Classes**
- **Shallow and Deep Initialization**
- **Overloading the Copy Constructor**
- **Shallow and Deep Assignment**
- **Overloading the Assignment**



# 16

This chapter looks at some of the problems that can arise when constructing and assigning incomplete objects (objects that use the heap or some other secondary resource). In particular, we will consider the concept of shallow and deep copies of objects. In a shallow copy only the object itself is copied; the associated heap storage is not. In a deep copy both the object and the associated storage are copied.

Shallow copies of incomplete objects invariably introduce serious problems into C++ programs. We see how shallow copies can be replaced with deep copies in initialization (by writing a copy constructor) and in assignment (by overloading the = operator).

## Copying Shallow Objects

```

class Shallow
{
    char* text;
    int x;
    int y;
public:
    Shallow (const char*, int, int);
    ~Shallow();
};

void Function(Shallow x)
{
    // body of function
}

```

```

int main()
{
    Shallow s("John", 27, 3);
    Function(s);
}

```

Consider what happens when a Shallow object s is passed by value to a global function and no copy constructor is provided for the class.

The semantics of parameter passing is defined as initialization. Therefore, when the function is called, the object s is used to initialize the parameter x. This initialization will be shallow because there is no copy constructor for the class.

What happens when the function completes? Since the x parameter is a local variable it must get destroyed. Destroying x will mean its destructor is called and hence the heap space deallocated. But s is still using the heap space!

Obviously, this is highly dangerous. Passing references provides a temporary solution, but how do we ensure this behavior is eliminated entirely.

## Assigning Shallow Objects

```

class Shallow
{
    char* text;
    int x;
    int y;
public:
    Shallow (const char*, int, int);
    Shallow ();
    ~Shallow();
};

Shallow Function()
{
    Shallow result;
    // body of function
    return result;
}

```

```

int main()
{
    Shallow s("John", 27, 3);
    s = Function ();
}

```

Copyright ©1994-2011 CRS Enterprises

213

Less obvious, but equally disastrous, is returning a shallow copy of an object from a function. The same forces are at work; the heap space gets deallocated while it is still in use by another object.

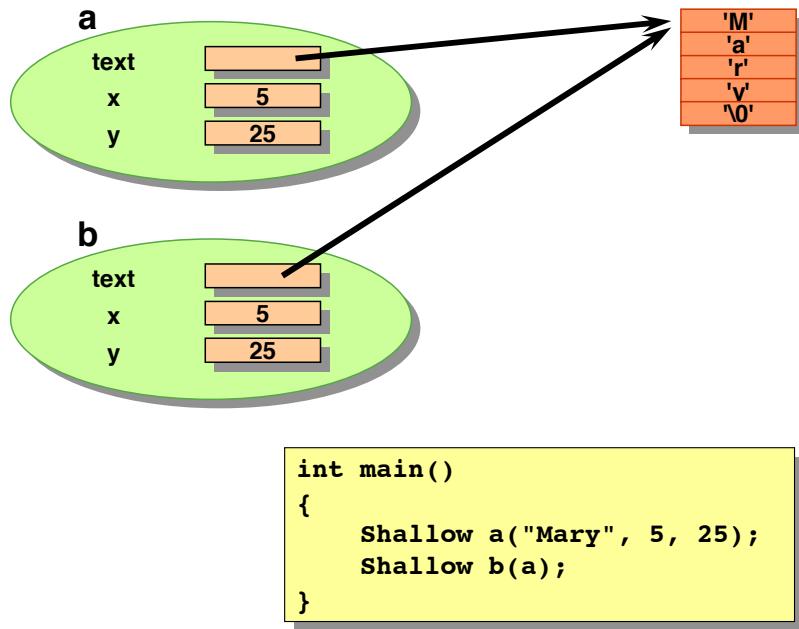
When the function is called it is allowed to create new objects. Such objects are local to the function and will be destroyed when we exit from the function. Let us suppose that the object `result` is properly constructed so that it does not share its heap space with any other object. The body of the function then modifies `result` as desired.

When the return statement is executed its semantics are also that of initialization. Hence,  
**return result;**

initializes an anonymous object in the function return area. This object will be a shallow copy of `result`. Thus the heap space used by `result` is now shared with the anonymous return object.

When the function completes, `result` is destroyed and its destructor called. The destructor deallocates the heap space still in use by the anonymous return object.

## Shallow Initialization



Copyright ©1994-2011 CRS Enterprises

214

When a shallow object is initialized with another object from the same class the compiler uses a special constructor called the copy constructor to perform the initialization. The compiler supplies a default copy constructor for each class, but as we have seen it is disastrous to rely on this constructor for incomplete objects.

Note that this problem only arises when the object to be initialized and the object used in the copy are from the same class. If object **b** was initialized as:

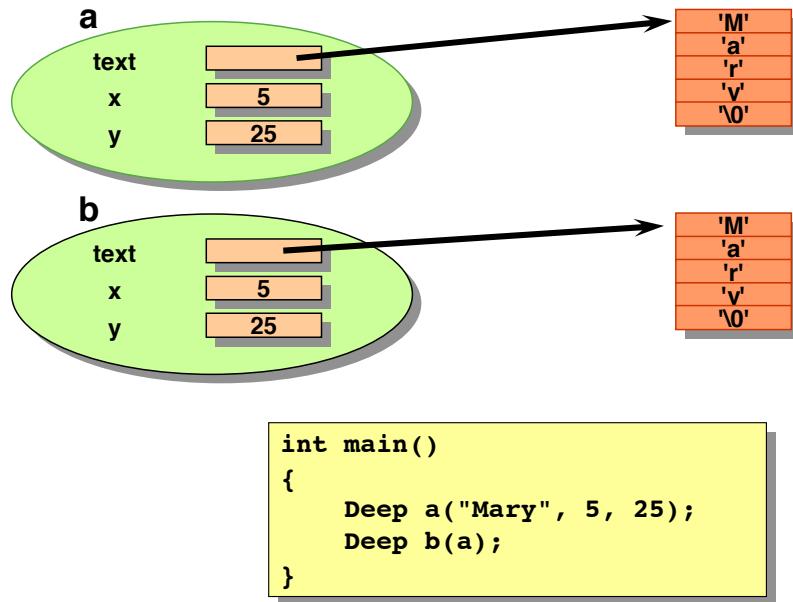
**Shallow b("blue", 5, 8);**

then the

**Shallow::Shallow(const char\*, int, int);**

constructor would be called. We have already written this constructor and it allocates heap storage correctly. What we need is to overload the copy constructor to provide similar functionality.

## Deep Initialization



Copyright ©1994-2011 CRS Enterprises

215

To perform a deep initialization a new constructor must be written, but which one? It looks as though we could write a constructor with one Deep input parameter:

**Deep::Deep(Deep)**

To see why this doesn't work consider creating a Deep object d1:

**Deep d1("green", 20, 30);**

No problem so far. However, when we attempt to create d2 from d1:

**Deep d2(d1);**

the Deep::Deep(Deep) constructor is called and a copy of d1 must be placed on the stack. But how do we construct the copy? We must call the Deep::Deep(Deep) constructor once again. Oops! this is a recursive loop.

# Overloading the Copy Constructor

*Deep.hpp*

```
class Deep
{
    char* text;
    int x;
    int y;
public:
    Deep(const Deep&);
};
```

*Deep.cpp*

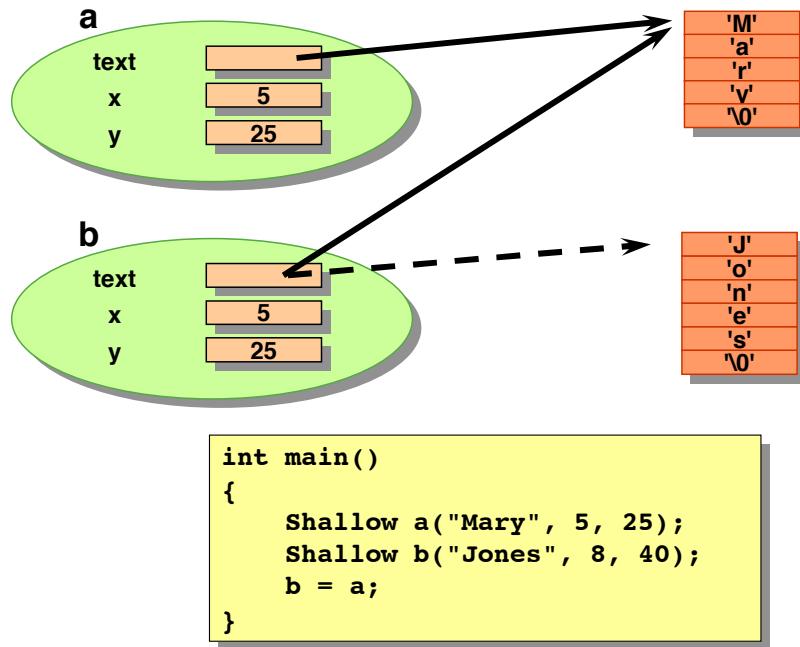
```
Deep::Deep (const Deep& object)
:
    x(object.x),
    y(object.y)
{
    int size = strlen(object.text) + 1;
    text = new char [size];
    strcpy(text, object.text);
}
```

The correct method of performing a deep initialization is to provide a copy constructor of the form:

**Deep::Deep(const Deep&)**

This copy constructor has a reference passed to it on the stack. References do not create copies of objects and therefore a further constructor is not called in this case! Hence no recursion.

## Shallow Assignment



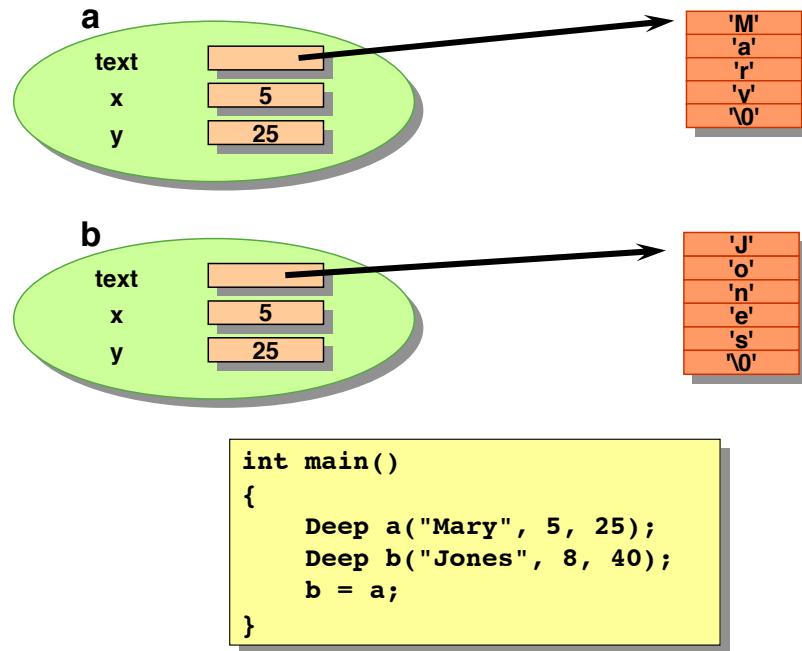
Copyright ©1994-2011 CRS Enterprises

217

When assignment is attempted between two incomplete objects, the compiler will always perform a shallow assignment of the objects unless you provide an overloaded assignment operator function.

With shallow assignment, the two objects share the same heap storage. We have the same disastrous situation: when either object's destructor is called the other object's heap space gets deallocated!

## Deep Assignment



Copyright ©1994-2011 CRS Enterprises

218

For incomplete objects it is essential to provide an overloaded assignment operator function for the class. The assignment operator function can be written such that a deep copy is made.

After a deep copy, objects a and b are independent from each other and no unwanted side effects ensue when either object is destroyed.

## Overloading the Assignment Operator

### *Deep.hpp*

```
class Deep
{
    char* text;
public:
    const Deep& operator=(const Deep&);
};
```

### *Deep.cpp*

```
const Deep& Deep::operator=(const Deep& copy)
{
    if (this == &copy) return copy;

    delete [ ] text;
    int size = strlen(copy.text) + 1;
    text = new char [size];
    strcpy(text, object.text);

    return *this;
}
```

This example shows how to write an overloaded assignment operator that performs a deep copy.

Before any assignment is attempted, it is wise to check that the client is not attempting to copy an object to itself. This is easily achieved:

**if (this == &object) return object;**

The x and y components of the object copied present no problem, but the text pointer must be handled carefully. The original heap space is first deleted:

**delete [ ] text;**

and then a new area of the heap allocated:

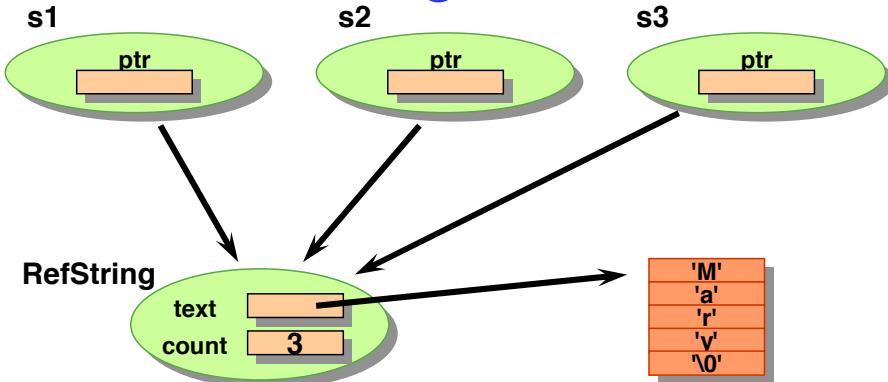
**int size = strlen(object.text);  
text = new char [size];**

Finally, the heap is initialized:

**strcpy(text, object.text);**

Note that the object to be copied is input to the function as a reference and returned from the function as a reference. This is more efficient than using copies. It also means that the object's constructor is not called!

## Reference Counting ...



```
int main()
{
    String s1("Blue");
    String s2(s1);
    String s3("Green");
    s3 = s1;
}
```

Copyright ©1994-2011 CRS Enterprises

220

We have seen that it is highly dangerous to rely on shallow copying and assignment semantics because the destructors deallocate shared secondary storage prematurely. This does not mean that we can't share storage, just that we have to arrange for the sharing to be carefully managed.

Reference counting is a popular technique whereby secondary storage can be safely shared. It involves the introduction of a new class to manage the heap space normally controlled by the incomplete objects.

In this example we create 3 String objects. When s1 is created its constructor creates a RefString to manage the heap space. The RefString also contains a reference count of the number of String objects currently sharing this space. The reference count is 1 at this stage.

When s2 is created it can share heap space with s1 simply by incrementing the RefString reference count to 2. Since s2 is created using a copy constructor it is easy to distinguish this situation from the former.

The String destructor decrements the reference count. Only if the reference count becomes zero should the RefString and heap space be removed. Full details are given overleaf.

## ... Reference Counting

### *String.cpp*

```

String::String(const char*)
{
    // create RefString
    // set count to 1
}
String::String(const String&)
{
    // increment count
}
const String& operator=(const String&)
{
    // decrement old count
    // delete RefString if zero
    // increment new count
}
String::~String()
{
    // decrement count
    // delete RefString if zero
}

```

Copyright ©1994-2011 CRS Enterprises

221

To implement reference counting we must define the constructors, destructor and operator= functions shown opposite in pseudo code.

The RefString is created when a String is created from a char\*.

The RefString is updated when a String is created via the copy constructor.

When a String is destroyed, its associated RefString is decremented. Only when the last String associated with the RefString is destroyed and the reference count goes to zero, is the RefString destroyed.

The assignment of String objects is essentially a combination of the destructor and copy constructor logic.

## Default Methods

### *Date.hpp*

```
class Date
{
public:
    Date();
    Date(const Date&);
    ~Date();
    const Date* operator&();
    const Date& operator=(const Date&);
};
```

### *Main.cpp*

```
int main()
{
    Date mayDay, christmas;
    Date today(mayDay);
    Date* ptr;

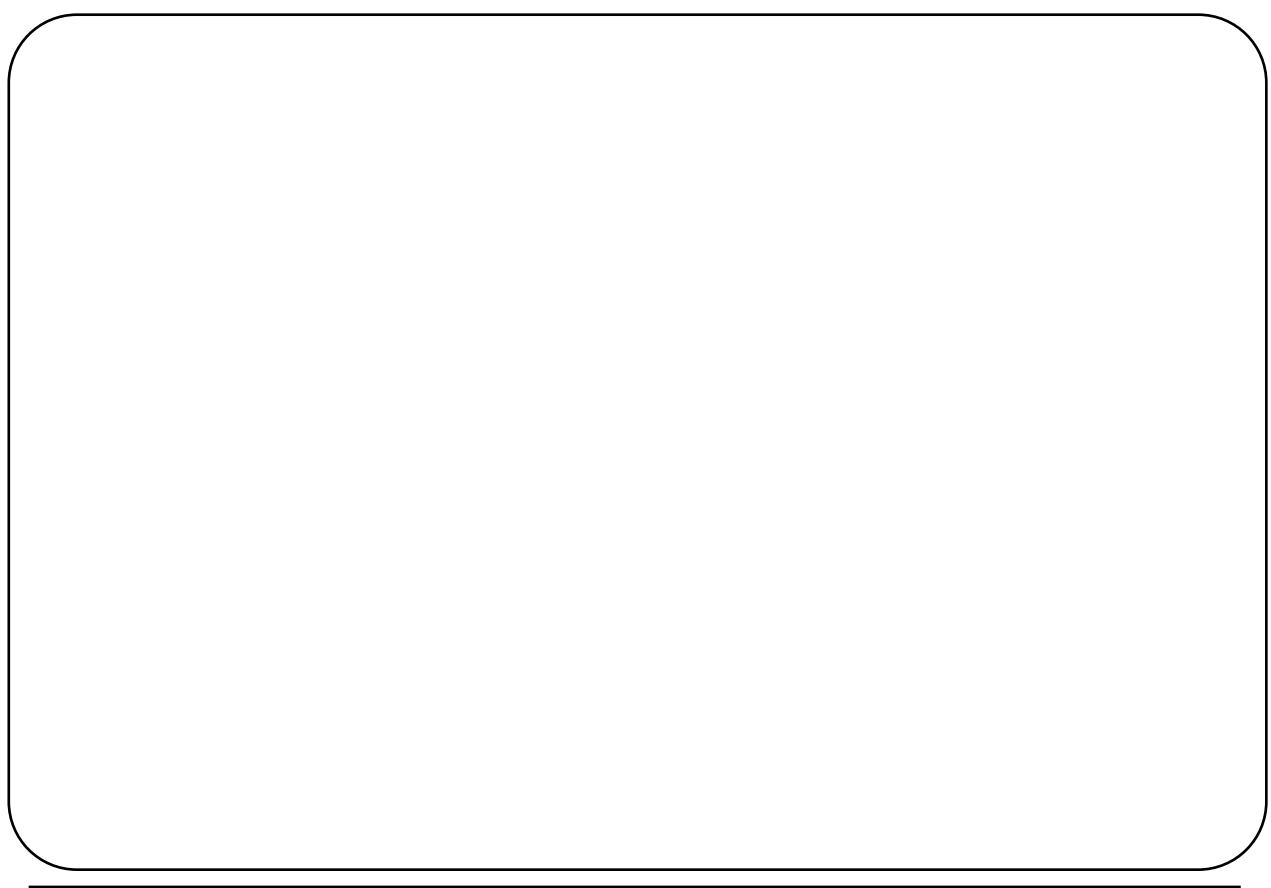
    ptr = &today;
    today = christmas;
}
```

Copyright ©1994-2011 CRS Enterprises

222

For every class you write, the compiler provides five default member functions as shown. For most classes these member functions are adequate. However, as we have seen, where classes contain incomplete objects it is mandatory to override the default copy constructor and assignment operator.

Note that the compiler will not generate its normal default constructor for the class if you provide any constructors of your own. This allows you to inhibit the creation of raw uninitialized objects.



# 17

## Standard Template Library

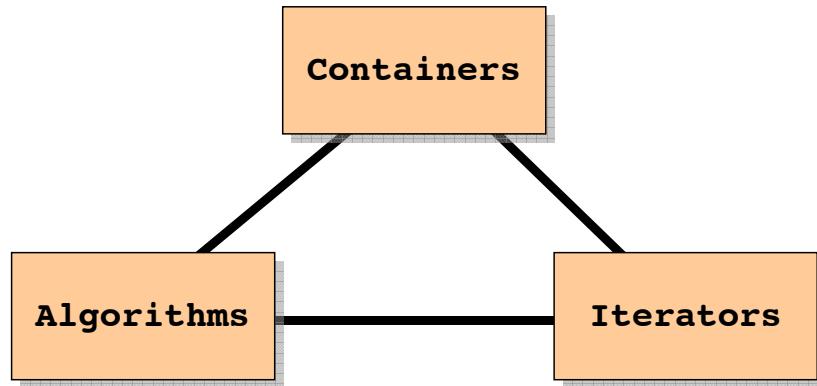
- **Containers**
  - **vector**
  - **list**
  - **map**
- **Algorithms**
- **Iterators**



**17**

This chapter investigates the Standard Template Library (STL). The STL provides an extensive set of data structures or containers and a large number of algorithms that operate on these containers.

## Overview of the STL



Copyright ©1994-2011 CRS Enterprises

226

The three components of the STL are a set of Containers, Algorithms that apply to the containers and Iterators that are used in the definition of these algorithms.

# Containers

- **Sequence containers:**
  - **vector**
  - **deque**
  - **list**
- **Container adaptors:**
  - **stack**
  - **queue**
  - **priority\_queue**
- **Associative containers:**
  - **set**
  - **multiset**
  - **map**
  - **multimap**
  - **bitset**

The STL contains sequence containers and associative containers. The standard sequence containers include vector, deque, and list. The standard associative containers are set, multiset, map, and multimap. There are also container adaptors queue, priority\_queue, and stack, that are containers with specific interface, using other containers as implementation.

# Algorithms

- **Many algorithms available**

- **copy**
- **swap**
- **transform**
- **replace**
- **replace\_if**
- **fill**
- **remove**
- **remove\_if**
- **unique**
- **reverse**
- **random\_shuffle**

A large number of algorithms to perform operations such as searching and sorting are provided in the STL, each implemented to require a certain level of iterator (and therefore will work on any container which provides an interface by iterators).

# Iterators

- **Input and output iterators**
  - most limited types of iterators
  - only sequential input or output operations
- **Forward iterators**
  - functionality of input and output iterators
  - limited to traversal in one direction
- **Bidirectional iterators**
  - can iterate in both directions
  - all standard containers support at least bidirectional iterators
- **Random access iterators**
  - implement functionalities of bidirectional iterators
  - can also access ranges non-sequentially

Input and output iterators are the most limited types of iterators, specialized in performing only sequential input or output operations.

Forward iterators have all the functionality of input and output iterators, although they are limited to one direction in which to iterate through a range.

Bidirectional iterators can be iterated through in both directions. All standard containers support at least bidirectional iterators types.

Random access iterators implement all the functionalities of bidirectional iterators and can also access ranges non-sequentially.

# Vector

- **Dynamic array**

- automatically expands when entries are added

```
#include <vector>

vector<string> collection;

collection.push_back("London");
collection.push_back("Madrid");
collection.push_back("New York");
collection.push_back("Tokyo");
collection.push_back("Rome");

for (unsigned i = 0; i < collection.size(); i++)
{
    cout << collection[i] << endl;
}
```

Use the vector class when you need a dynamic array. The array will automatically resize as elements are added. The underlying implementation takes care of the resizing, but typically this will reserve additional space to allow the array to expand. If the additional space is exhausted, the implementation may relocate the array and double its size. To avoid resizing, a constructor is provided that allows the caller to specify the initial size of the array.

Note that the [ ] operator is overloaded as a convenience to allow the array to be used like a C array. Unfortunately, the vector class does not provide bounds checking, although it is a simple matter to derive a class from vector and provide bounds checking via an overloaded [ ] operator function.

## Vector using Iterators

- **Iterator is a nested class of vector**
  - random access iterator
  - overloaded operators `++ -- * == !=`

```
vector<string> collection;

collection.push_back("London");
collection.push_back("Madrid");
collection.push_back("New York");
collection.push_back("Tokyo");
collection.push_back("Rome");

vector<string>::iterator i;

for (i = collection.begin(); i != collection.end() ; ++i)
{
    cout << *i << endl;
}
```

Copyright ©1994-2011 CRS Enterprises

231

In order to perform algorithms with the vector class you will need to use iterators. The vector class will supply a random access iterator. A simple example of using such an iterator is shown above.

## List using Iterators

- Simple to change vector to list
  - illustrates similarity of methods

```
#include <list>

list<string> collection;

collection.push_back("London");
collection.push_back("Madrid");
collection.push_back("New York");
collection.push_back("Tokyo");
collection.push_back("Rome");

list<string>::iterator i;

for (i = collection.begin(); i != collection.end() ; ++i)
{
    cout << *i << endl;
}
```

Copyright ©1994-2011 CRS Enterprises

232

The vector and list classes are designed to have a similar set of methods. Obviously, the implementations of the two classes are quite different; the list class implements a linked list and the vector an array.

The similarity of methods make it easy to convert our previous example from a vector implementation to a list implementation.

## Maps ...

```
std::map <key_type, data_type, [comparison_function]>
```

- **inserting**

```
map<string, Point*> mymap;
pair<string, Point*> mypair;
mypair.first = "point-1";
mypair.second = new Point(3,4);
mymap.insert(mypair);
```

- **inserting**

```
mymap.insert(make_pair("point-2", new Point(4,5)));
```

- **inserting**

```
mymap[ "point-3" ] = new Point(5,6);
```

The Map class provides an insert method to add or modify entries. The insert method is overloaded, but the one shown above uses a pair object to specify the inserted element.

The pair class is a simple structure:

```
template <typename T1, typename T2>
struct pair
{
    T1 first;
    T2 second;
}
```

As you can see, using a pair is a little cumbersome, so the [ ] operator function (which calls insert) is often used in practice. Alternatively, use the make\_pair function to set up the key-value pair.

## ... Maps

- **finding**

```
map<string, Point*> mymap;
map<string, Point*>::iterator i;
i = mymap.find("point-3");
Point* ptr;
string key = i->first;
ptr = i->second;
```

- **finding**

```
ptr = mymap["point-2"];
```

- **erasing**

```
mymap.erase("point-2");
```

To look up a key in a map you can use the find method. The find method returns an iterator that points to a pair object. Again the the [ ] operator function is a convenience to simplify accessing the map.

To remove a key from a map use the erase method.

## ... Maps

- **vendors decide on implementation**
  - hash, tree
- **no two elements have the same key**
  - unique keys
- **inserting**
  - does not invalidate existing iterators
- **erasing**
  - does not invalidate existing iterators
  - except for iterators pointing to erased element
- **multimap**
  - like map, but can have duplicate keys

Vendors decide on the implementation of the Map, but typically will use a hash or tree implementation.

Maps have unique keys: no two elements have the same key (unique keys). Alternatively, if you need elements with duplicate keys you can use a multimap.

Map has the important property that inserting a new element into a map does not invalidate iterators that point to existing elements. Erasing an element from a map also does not invalidate any iterators, except, of course, for iterators that actually point to the element that is being erased.

## Algorithms - find\_if

```
bool LessThan21K(const Employee& e)
{
    if (e.GetSalary() < 21000)
        return true;
    else
        return false;
}
```

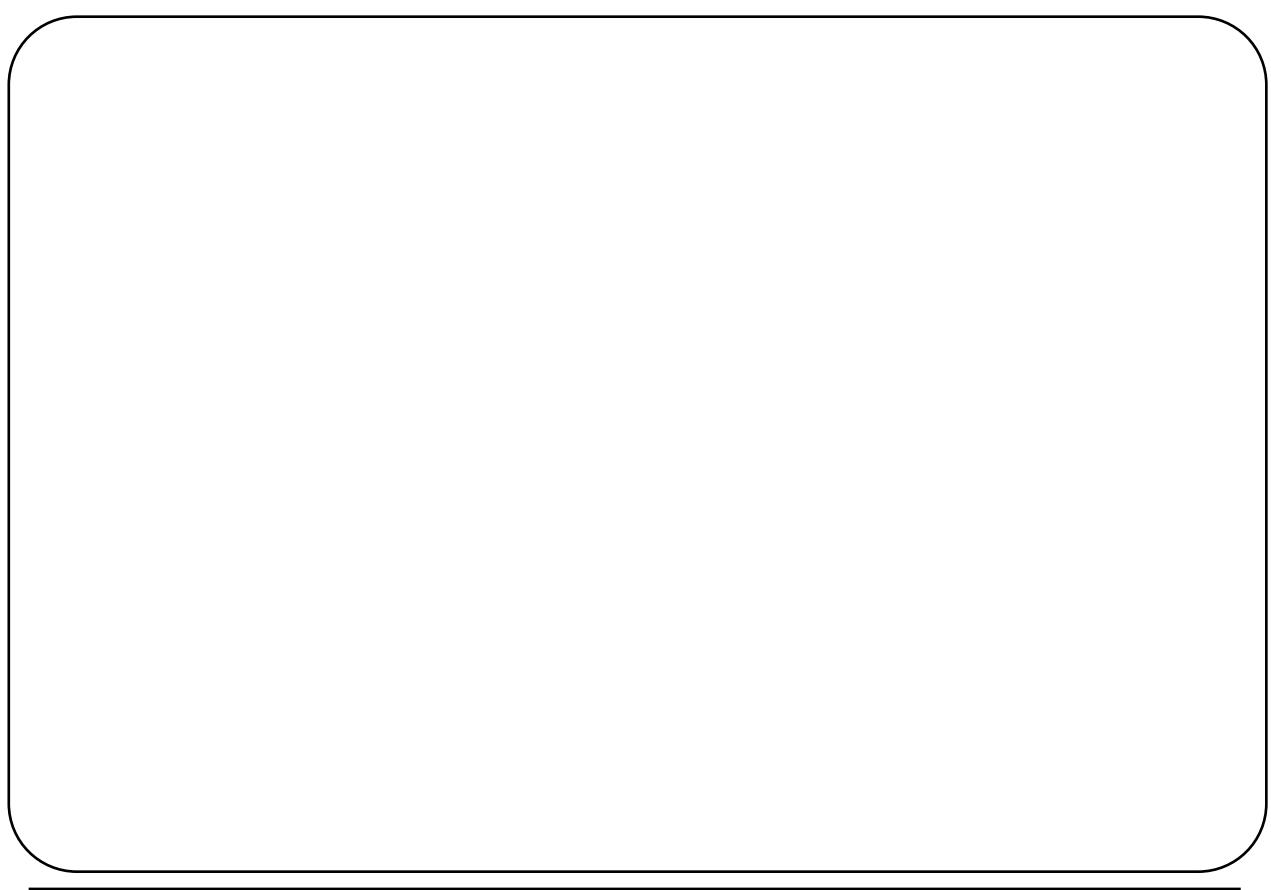
```
list<Employee> collection;
list<Employee>::iterator i;
i = find_if(collection.begin(),
            collection.end(),
            LessThan21K);
if(i != collection.end())
    i->Print();
else
    cout << "not found" << endl;
```

Copyright ©1994-2011 CRS Enterprises

236

The `find_if` algorithm is typical of those provided by the STL. The algorithm takes three parameters; the first two parameters are iterators which define the start and end of the range to be searched. The third parameter is a function pointer (or functor) that points to a predicate function (a function that returns true or false) which acts a callback mechanism.

The implementation will repeatedly callback on the predicate function with elements of the collection until either the function returns true or the end of the collection is reached. The algorithm returns an iterator to the selected element in the former case and the end iterator in the latter case.



# 18

## Inheritance

- **Public Inheritance**
- **Comparison with Aggregation**
- **Constructor Issues**
- **Protected and Private Inheritance**
- **Multiple Inheritance**



# 18

New classes can be derived from a base class using inheritance. Inheritance differs from aggregation in that the new class inherits all the attributes and methods from the original class. The new class can define additional attributes and methods.

C++ provides the protected access level for attributes and methods used in inheritance. Protected attributes and methods are directly accessible by derived classes, but inaccessible to client programs.

There are two types of inheritance, single inheritance and multiple inheritance. We will concentrate on single inheritance in this section. Single inheritance can be subdivided into public, protected and private inheritance. Public inheritance is by far the most common form of single inheritance.

## Public Inheritance

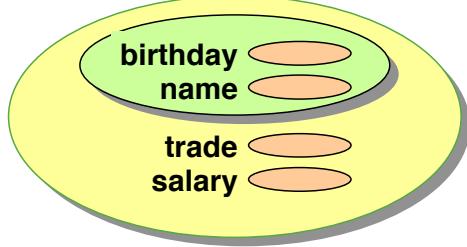
```
class Person
{
private:
    Date birthday;
    String name;
public:
    GetName();
    GetAge();
};
```

Person paul;



```
class Worker : public Person
{
private:
    String trade;
    Money salary;
public:
    ChangeTrade();
    PrintSalary();
};
```

Worker nurse;



Copyright ©1994-2011 CRS Enterprises

240

Consider the Person class opposite. This is a base class for the derived class Worker. The base class has attributes birthday and name and methods GetName() and GetAge().

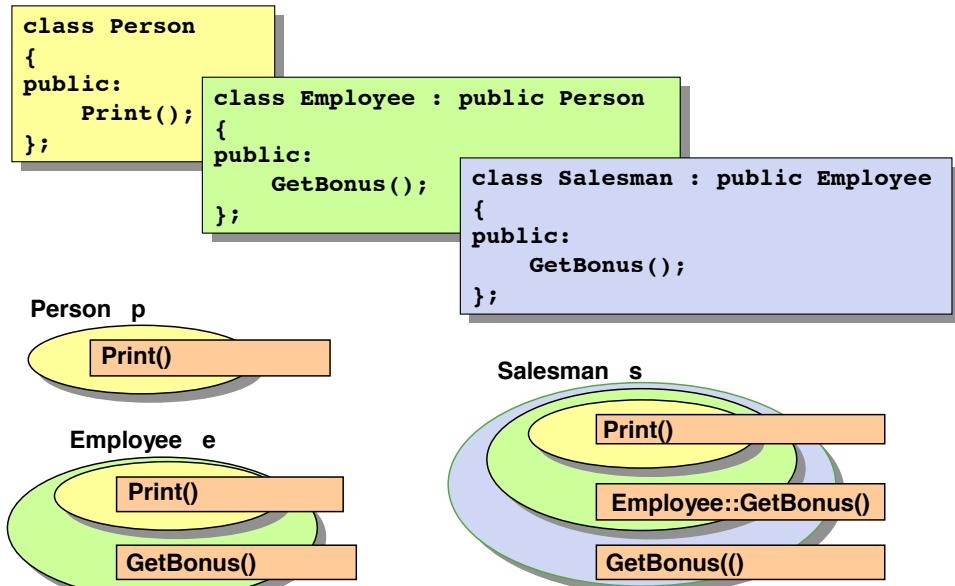
The derived class defines the additional attributes and methods over and above those already provided by the base class. Thus Worker defines additional data components trade and salary and member functions ChangeTrade() and PrintSalary().

The derived class inherits all the base class's attributes and methods using the syntax:

**class Derived : public Base**

The derived class now has attributes birthday, name, trade, salary and methods GetName(), GetAge(), ChangeTrade(), PrintSalary().

## Inheriting Methods



Copyright ©1994-2011 CRS Enterprises

241

Member functions of a base class are inherited by all classes in an inheritance hierarchy. In the example opposite the Print() member function was originally defined in the Person class, but the two derived classes, Employee and Salesman also inherit the same member function. Similarly, the GetBonus() member function was originally defined in class Employee, but is also available to class Salesman.

Derived classes can also modify the behavior of the member functions they inherit by providing an alternative member function with the same name. Consider the class Salesman. This class inherits the GetBonus() member function from class Employee, but also defines its own version of GetBonus(). If a GetBonus() message is sent to an object a of the Salesman class then the new version will be used.

```

Salesman jim;
jim.GetBonus();

```

The inherited method is still available. To send the inherited message you must prefix the member function with its class name:

```

Salesman jim;
jim.Employee::GetBonus()

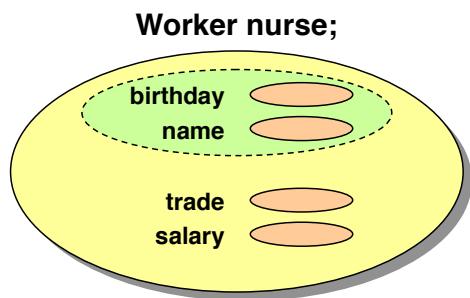
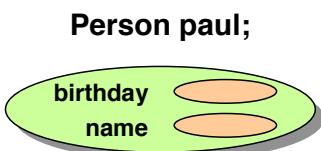
```

Note that although Employee::GetBonus() and Salesman::GetBonus() appear to be overloaded functions, this is not the case. Remember that overloaded functions must have distinct signatures. Functions with the same name in an inheritance hierarchy are called polymorphic functions.

## Protected Access

```
class Person
{
protected:
    Date birthday;
    String name;
public:
    GetName();
    GetAge();
};
```

```
class Worker : public Person
{
private:
    String trade;
    Money salary;
public:
    ChangeTrade();
    PrintSalary();
};
```



Copyright ©1994-2011 CRS Enterprises

242

Attributes and methods that are declared to be private in a base class remain inaccessible even to derived classes. In the example opposite, the derived class is not allowed to perform the assignments:

```
birthday = 10_september_1998;
name = steven_morris;
```

because birthday and name are declared private in the base class. This is rather strange. Although birthday and name were declared private in the base class, any object in the derived class actually contains storage for birthday and name. This means that objects in the derived class are denied access to their own data!

If attributes and methods are declared as protected rather than private in a base class, then derived classes are allowed direct access to them. Client programs are still denied access to protected attributes and methods as was the case with private attributes and methods.

From an object oriented viewpoint, the use of protected components violates encapsulation and should be avoided. This feature has been included in C++ to improve efficiency.

## Access Levels

	base class	derived class	client code
public	✓	✓	✓
protected	✓	✓	✗
private	✓	✗	✗

There are now three different access levels available for declaring attributes and methods within a class.

Public attributes and methods can be accessed directly by everything.

Protected attributes and methods can be accessed directly by all derived classes, but not client code.

Private attributes and methods are only accessible to the base class.

## Comparison with Aggregation

Inheritance	Aggregation
<i>Employee IS A Person</i>	<i>Car HAS A Wheels</i>
<b>Employee SUBSTITUTES for Person</b>	<b>NO SUBSTITUTION rule</b>
<b>ONE sub-object</b>	<b>SEVERAL sub-objects</b>
<b>STRICT</b> <i>Data Encapsulation for outer layer</i>	<b>STRICT</b> <i>Data Encapsulation for outer layer</i>
<b>OPTIONAL</b> <i>Data Encapsulation for inner layers</i>	<b>STRICT</b> <i>Data Encapsulation for inner layers</i>
<i>Member Functions INHERITED from inner layers</i>	<i>Member Functions NOT INHERITED from inner layers</i>

Copyright ©1994-2011 CRS Enterprises

244

Both inheritance and aggregation can be used to create new classes from old. It is vitally important to recognize when to use inheritance and when to use aggregation.

The most important distinction between inheritance and aggregation is in the IS A relationship. Only use inheritance if the new class is related to the original class by an IS A relationship. This in turn leads to the fundamental substitution rule: objects of derived classes may safely be substituted for base class objects. Thus the global function

**PrintAnyObject(Person&)**

can take Employees and Person objects as its argument.

Since every derived object contains only one base sub-object, inheritance cannot be used when you want to build aggregates of sub-objects as in the Car and Wheels example.

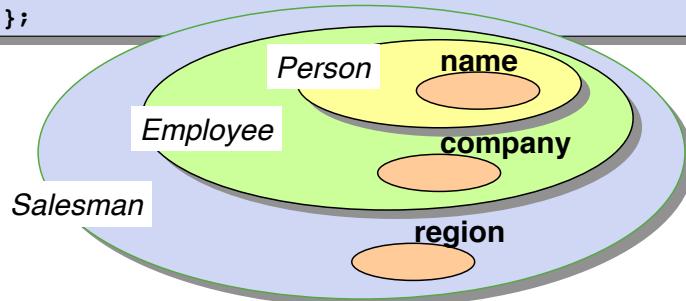
Finally, note that protected inheritance violates encapsulation at inner levels (for efficiency reasons). Stick with strict data encapsulation if you want to be fully Object Oriented compliant.

## Constructor Issues ...

```
class Person {
private:
    String name;
public:
    Person(String& n);
};
```

```
class Employee : public Person {
private:
    String company;
public:
    Employee(String& n, String& c);
};
```

```
class Salesman : public Employee {
private:
    String region;
public:
    Employee(String& n, String& c, String& r);
};
```



Copyright ©1994-2011 CRS Enterprises

245

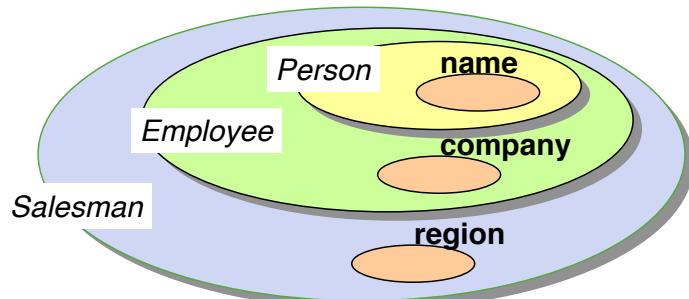
In the class hierarchy opposite we have three classes Person, Employee and Salesman; each class is provided with its own constructor. Note that objects in class Salesman have a region attribute and an Employee sub-object, those in class Employee have a company attribute and a Person sub-object , while those in class Person have but a name attribute. This is reflected in the parameters required by the respective constructors.

## ... Constructor Issues

```

Person::Person(String
& n) :
    name(n)
}
{} Employee::Employee(String& n, String& c) :
    Person(n),
    company(c)
}
{} Salesman::Salesman(String& n, String& c, String& r) :
    Employee(n, c),
    region(r)
}

```



Copyright ©1994-2011 CRS Enterprises

246

When an object of class Salesman is declared:

**Salesman jim(Jim,LSI,South);**

the constructor **Salesman::Salesman(String&, String&, String&)** is called. Note that the initialization list in this constructor initializes the locally defined component: **name(n)**, but invokes the constructor

**Employee::Employee(String&, String&)**

to initialize the rest of the object.

The class Employee constructor in turn initializes its locally defined component **company(c)**, but invokes the constructor

**Person::Person(String&)**

to initialize the remainder of the object.

Note that the class Salesman constructor does not call the class Person constructor directly; this would violate encapsulation!

## Protected/Private Inheritance

- **Not Standard Inheritance**
  - inheritance of implementation
  - more like aggregation
- **Substitution Rule does not work**
  - derived classes remove methods and attributes
- **Example**
  - creating a Stack from a List

C++ supports two additional types of inheritance: protected and private. These forms of inheritance are not standard inheritance, but more akin to a form of aggregation. More formally, these forms are called inheritance of implementation.

The key point about these forms of inheritance is that methods can be hidden by the deriving class using the protected or private qualifier. This means that derive class objects do not exhibit the same public interface as their base class counterparts, which in turn means they cannot substitute for base class objects.

Thus the substitution rule does not apply for this type of inheritance. This in turn means that polymorphism (see next chapter) is not applicable to these forms of inheritance.

Inheritance of implementation is usually used when a class wishes to publish a different interface from the base class, but would like to utilize the base class implementation.

For example a List class and a Stack class should behave differently in the way items are added and removed - the List can insert and delete anywhere, but the Stack only at its top. Using private inheritance, the Stack class could inherit the implementation of a List class, but restrict how these methods are used by defining its own public interface (Push and Pop) and making the inherited methods private.

## Multiple Inheritance

```
class Trip
{
    Distance kilometers;
    String Destination;
public:
    void Hotel(String&);
};
```

```
class Business
{
private:
    String project;
public:
    void ManHours(Time&);
};
```

```
class BusinessTrip : public Business, public Trip
{
public:
    void Expenses(Money&);
};
```

Multiple inheritance is where a derived class inherits from two or more base classes. In this example:

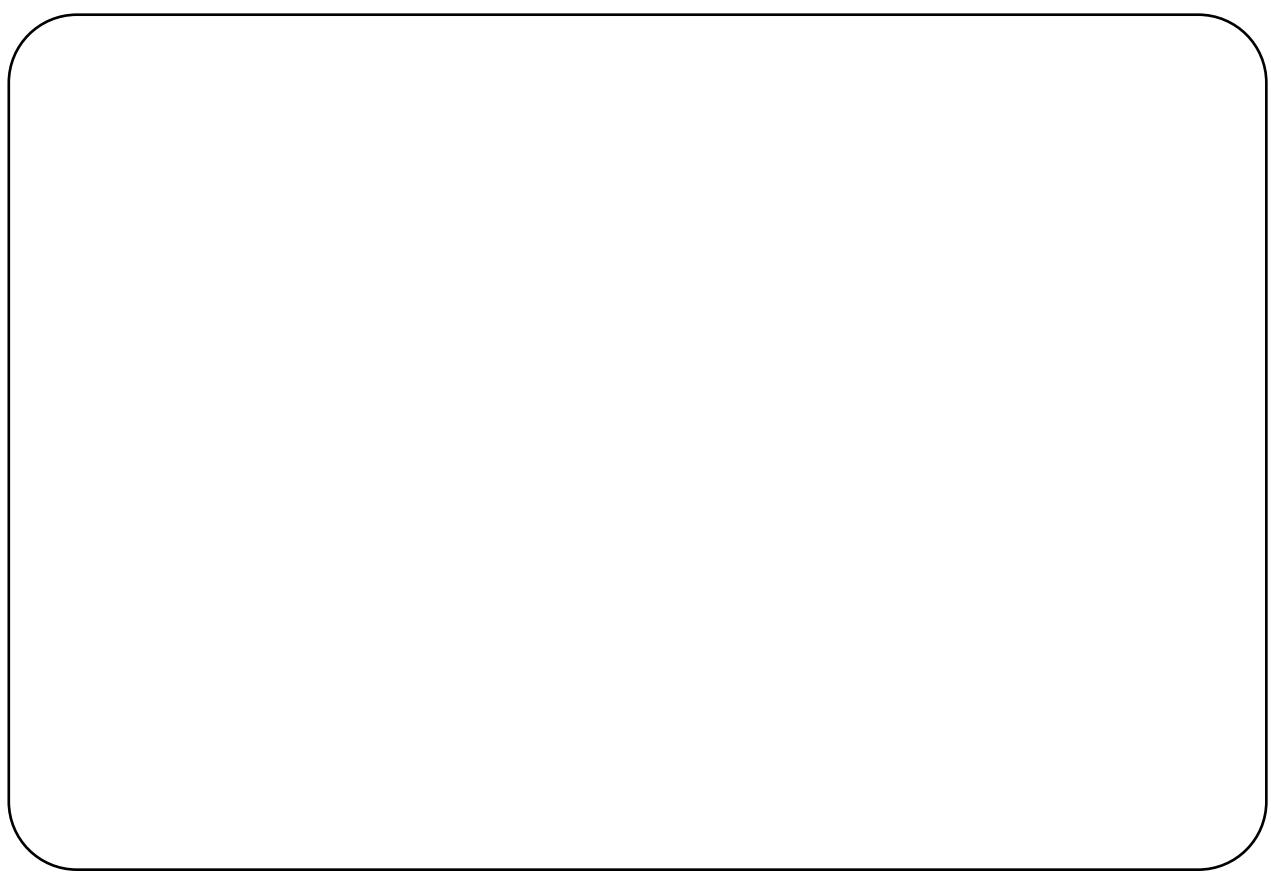
**class BusinessTrip : public Business, public Trip**

allows the derived class to inherit all attributes and methods from the two base classes. This is really useful. Multiple inheritance is an extremely powerful Object Oriented concept.

Clearly a BusinessTrip object IS A Business object and at the same time IS A Trip object. We can reap the benefits from both base classes using multiple inheritance.

It should be stressed however that the use of multiple inheritance is a highly emotive one; designs using multiple inheritance can become very complicated and ambiguities can arise. What happens, for instance, if each of the base classes are derived classes from a common ancestor class. This will produce duplicate attributes in our class. Worse still, we will have duplicate constructors and duplicate methods. What happens if the different base classes contain attributes with identical names?

C++ can resolve such ambiguities, but does so in a cumbersome manner. For the details attend the Advanced C++ course!



# 19

## Polymorphism

- **What is Polymorphism?**
- **Virtual Table Mechanism**
- **Virtual Functions**
- **Dynamic Binding**



# 19

In this chapter we will investigate inheritance in more detail. In particular we will look at the concepts of polymorphism and dynamic binding and investigate the mechanisms used by the C++ compiler to implement these paradigms.

We will also look at abstract base classes with pure virtual functions and discuss virtual destructors.

## Substitution Rule

```
class Person
{ };
class Employee : public Person
{ };
class Manager : public Employee
{ };
```

```
int main()
{
    Person john;
    Employee worker;
    Manager boss;

    Person* ptr;

    ptr = &john;
    ptr = &worker;
    ptr = &boss;
}
```

Copyright ©1994-2011 CRS Enterprises

252

In our previous discussion on inheritance we have been careful to avoid using pointers to objects. Special considerations come into play when we define a pointer to a class that is part of an inheritance hierarchy.

Consider the declaration

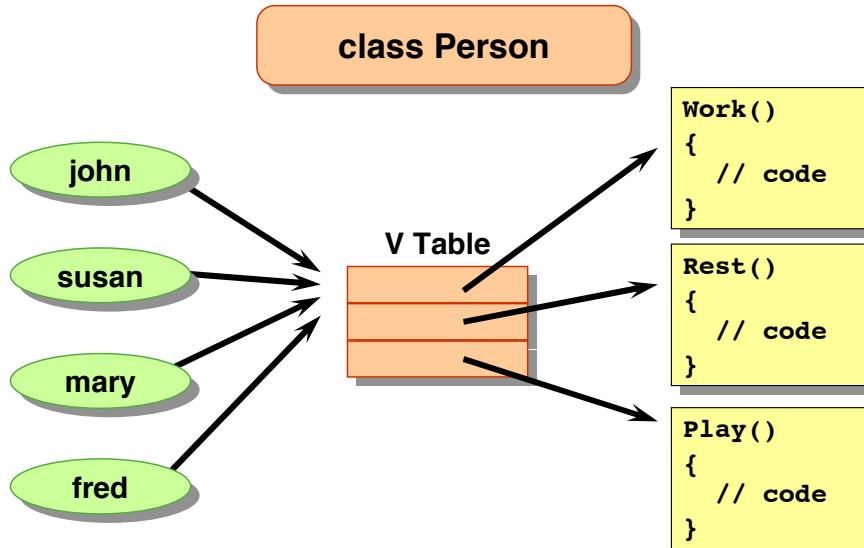
**Person\* ptr;**

In C this would define a pointer to an object of class Person, but in C++ it defines a pointer to an object belonging to any of the classes in Person class hierarchy. This is because of the substitution rule:

**In any inheritance hierarchy, any derived class object may be substituted for a base class object wherever it would be legal to use a base class object.**

The substitution rule is a restatement of the fact that a derived object IS A kind of base object. The substitution rule forms the basis of polymorphism.

## The V-Table



Copyright ©1994-2011 CRS Enterprises

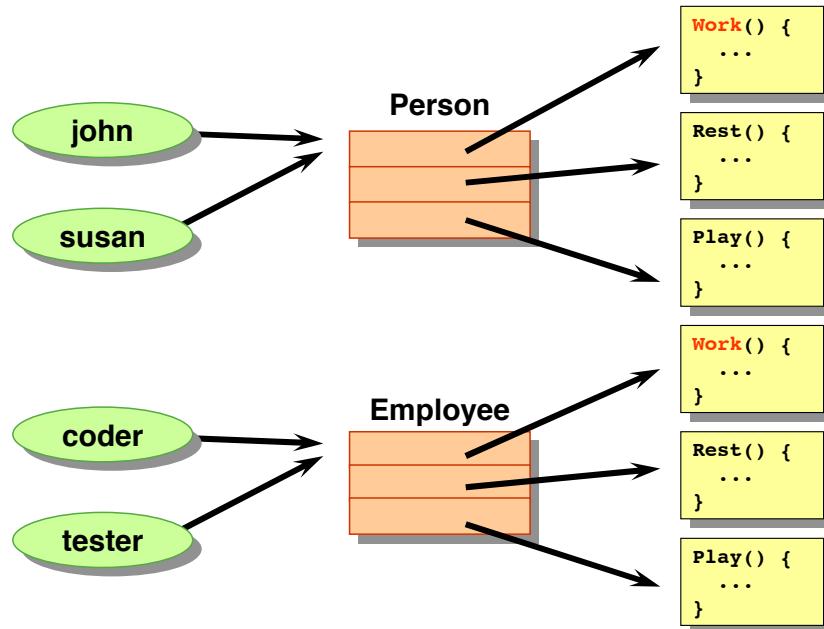
253

Consider a class Person with 3 methods Work(), Rest() and Play() and 4 objects john, susan, mary and fred.

Under certain circumstances the C++ compiler embeds an implicit pointer at the end of each object which points at the class's V-Table. The V-Table contains a set of pointers to the methods and can therefore be used at run time to determine the addresses of these methods. Each class has a single V-Table.

With objects, the compiler can bind objects to messages at compile time and the V-Table mechanism is not needed. However, when pointers are used to access objects, the compiler resolves object to message bindings at run time using the V-Table.

## Multiple V-Tables

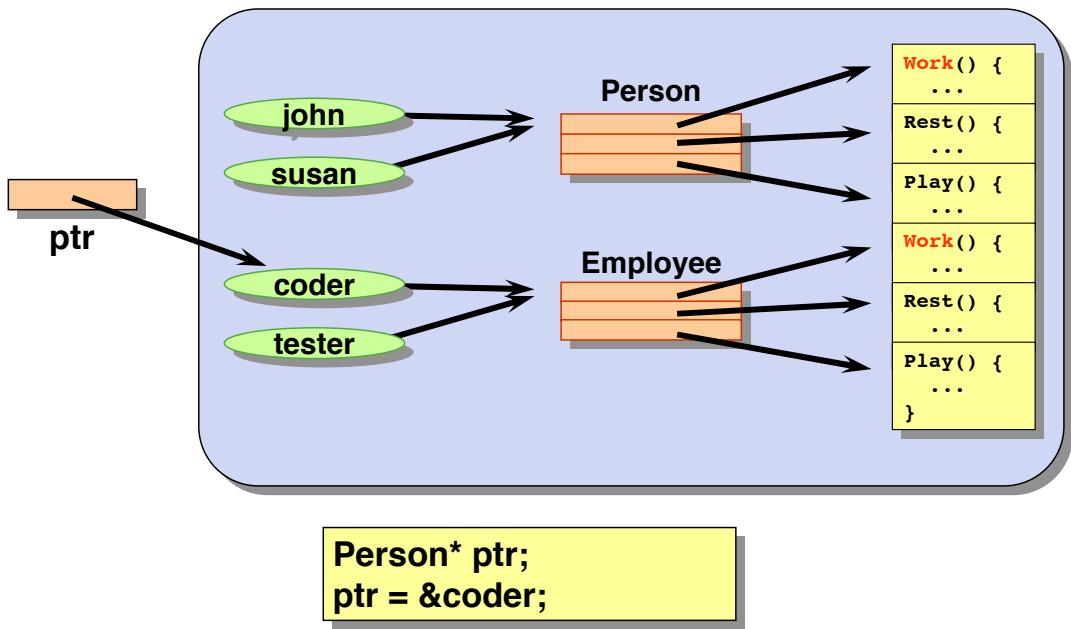


Copyright ©1994-2011 CRS Enterprises

254

Now suppose a derived class (**Employee**) is added to an inheritance hierarchy which redefines each of the methods in the base class (**Person**). The C++ compiler will generate a separate V-Table for the new class that points to these new methods. The V-Table for the new class will be exactly the same size as that for the base class, but will contain different function addresses.

## Base Class Pointers



Copyright ©1994-2011 CRS Enterprises

255

When a pointer to the base class is declared, as in

**Person\* ptr;**

the compiler does not know to which class ptr is pointing, since ptr can point not just to objects in the base class but also to objects in every derived class. Effectively, pointers point at class hierarchies rather than individual classes.

When a pointer is dereferenced as in

**ptr->Rest()**

the compiler generates code that will work irrespective of which class ptr actually points at. This run time code performs the tasks:

**dereference the pointer to select the object**

**dereference the embedded pointer at the end of object to locate the V-Table**

**use the 2nd function pointer to local the Rest method**

**invoke the method**

The scheme relies on the fact that each V-Table is the same size and contains pointers to the polymorphic methods in the same sequence.

## Virtual Functions

```
class Person
{
public:
    virtual void Work();
    virtual void Rest();
    virtual void Play();
};

class Employee : public Person
{
public:
    virtual void Work();
    virtual void Rest();
    virtual void Play();
};

class Manager : public Employee
{
public:
    virtual void Work();
    virtual void Rest();
    virtual void Play();
};
```

Copyright ©1994-2011 CRS Enterprises

256

Not all methods have entries in the V-Table. The compiler only includes polymorphic methods. The class designer chooses which methods are to be polymorphic by qualifying the methods with the `virtual` qualifier.

If a method is polymorphic in the base class it is automatically polymorphic in all derived classes (whether qualified as `virtual` or not). Each derived class must have exactly the same number of polymorphic methods as the base class.

## Dynamic Binding

```
int main()
{
    Person john;
    Employee worker;
    Manager boss;

    FeedPeople(&john);
    FeedPeople(&worker);
    FeedPeople(&boss);
}
```

```
void FeedPeople(Person* ptr)
{
    ptr->Eat();
}
```

Copyright ©1994-2011 CRS Enterprises

257

Consider the function

```
void FeedPerson(Person* ptr)
{
    ptr->Eat();
}
```

A pointer to a Person object is passed to the function and an Eat message is then sent to this object. Which Eat method is called?

Remember that ptr can point to Person, Employee or Manager objects. If ptr really points to a Person object then we intend ptr->Eat() to mean

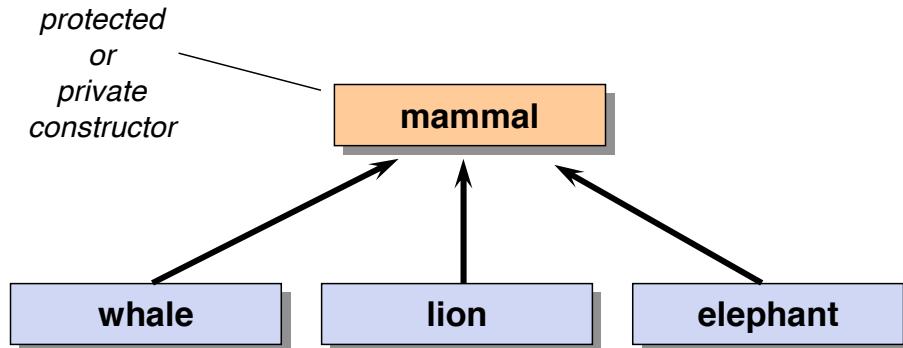
**ptr->Person::Eat()**

However, if ptr points to an Employee object then we intend ptr->Eat() to mean  
**ptr->Employee::Eat()**

and similarly for Manager objects.

The compiler delays the binding of object to method (\*ptr to Eat) until run time and hence this is called dynamic binding.

## Abstract Classes



Copyright ©1994-2011 CRS Enterprises

258

Base classes contain data and methods that are inherited by all derived classes. In other words they contain the common elements of the derived classes. Sometimes base classes are created for the sole purpose of encapsulating these common elements and there is no intention of instantiating the class.

For example, we might define a **mammal** class as the base class for the **whale**, **lion** and **elephant** classes. The **mammal** class serves the useful purpose of defining the elements common to the **whale**, **lion** and **elephant** classes, but we will never create a **mammal** object.

Such classes are called **abstract classes**. We can ensure the class has no objects by making its constructors inaccessible to the client code; we declare all constructors for an abstract class as **protected**!

## Pure Virtual Functions

```

class Person
{
protected:
    Person();
public:
    virtual void Work()      = 0;
    virtual void Rest()      = 0;
    virtual void Play()      = 0;
};

class Employee : public Person
{
public:
    Employee();
    virtual void Work();
    virtual void Rest();
    virtual void Play();
};

```

Copyright ©1994-2011 CRS Enterprises

259

If a base class is an abstract class, it seems pointless defining methods for the class. However, we must define virtual methods for all classes in the inheritance hierarchy to ensure all V-Tables are the same size. To avoid writing unnecessary methods, we can define pure virtual functions as in

**virtual void Work() = 0;**

This method is superior to defining Work() as

**virtual void Work() {}**

since the compiler will ensure a virtual method is never called. If a class has one or more pure virtual functions the compiler will refuse to instantiate the class. In other words, this is a second method of defining an abstract class.

## Virtual Destructors

```

class Person
{
protected:
    Person();
public:
    virtual void Work()      = 0;
    virtual void Rest()       = 0;
    virtual void Play()       = 0;
    virtual ~Person();
};

class Employee : public Person
{
public:
    Employee();
    virtual void Work();
    virtual void Rest();
    virtual void Play();
    virtual void ~Employee();
};

```

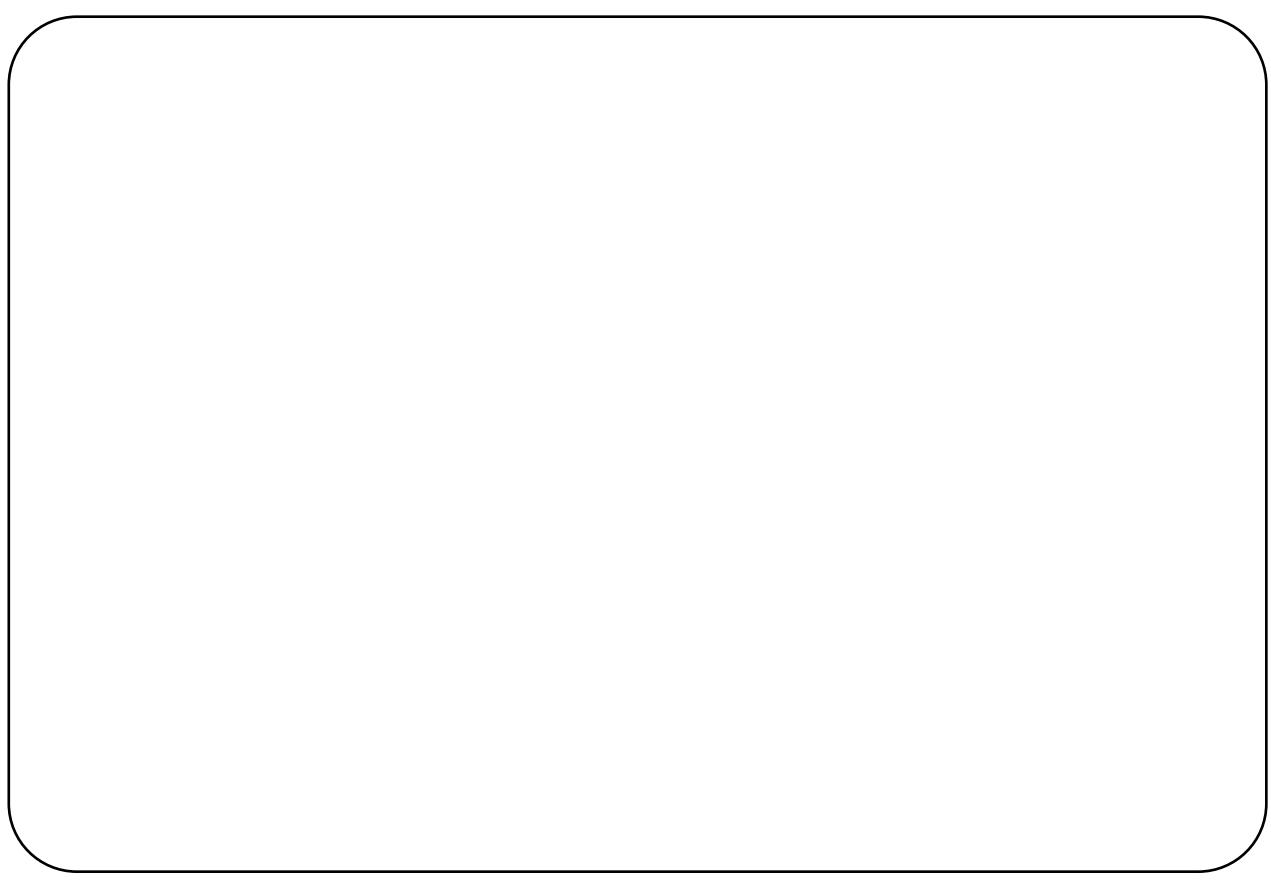
Copyright ©1994-2011 CRS Enterprises

260

Finally, note that it is usually a good idea to make a destructor virtual. Virtual destructors must be used if the classes in the inheritance hierarchy are incomplete (contain pointers).

In such cases, if the destructor is not virtual then secondary resources will not be cleaned up properly; the compiler will resolve references at compile time and will always call the base destructor. By declaring the destructor virtual, we ensure dynamic binding and hence the correct destructor will be called.

A final point: although destructors are often declared virtual, constructors are never allowed to be declared virtual.



# 20

## More on Operator Overloading

- **Casts Using Constructors**
- **Casts Using Operators**
- **Overloading [ ]**
- **Overloading ( )**
- **Smart Pointers**



# 20

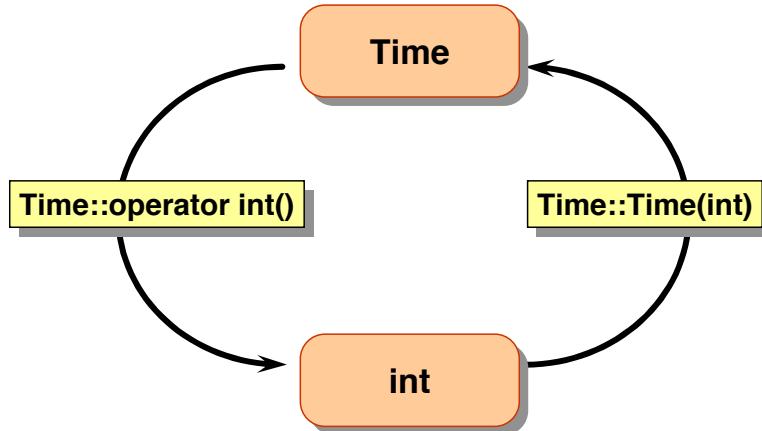
In this section we take a look at user defined cast operators and how to overload the [ ], () and -> operators.

There are two types of cast operators in C++: casts which create temporary objects and casts which convert objects to built in types or different classes. Constructors are used to create temporary objects. Special operator functions must be defined for casts which convert objects.

The [ ] and () operators are used to generalize the C array concept. C++ implementations can easily add bounds checking and define associative arrays.

Overloading the -> operator allows us to define smart pointers. These pointers can be made smart enough to refuse to dereference themselves when they are null. Smart pointers have many other uses outside the scope of this treatment.

## Casts ...



Copyright ©1994-2011 CRS Enterprises

264

C++ allows you to define two types of cast between a class and a built in data type. A cast is regarded as construction when a class object is created from a built in type and conversion when the object is broken down into the built in type.

The constructor

`Time::Time(int)`

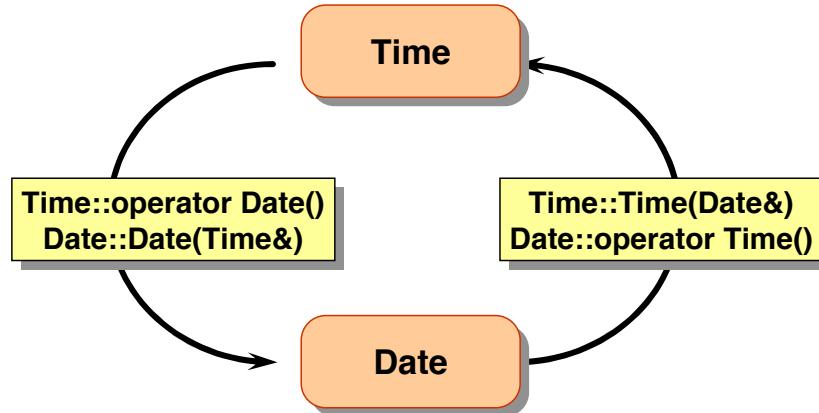
builds a `Time` object from an `int`.

The conversion

`Time::operator int()`

converts a `Time` object into an `int`.

## ... Casts



Copyright ©1994-2011 CRS Enterprises

265

The situation is a little more complicated when we consider casts between user defined classes. If we simply extrapolate from the discussion on int to Time conversion functions we would expect that

**Time::Time(Date&)**

would construct a Time object from a Date and

**Date::operator Date()**

would convert a Time into a Date object. This is indeed the case. However, this discussion is biased towards the Time class. If we look at things from the Date class we see that we could define

**Date::Date(Time&)**

to construct a Date object from a Time and

**Date::operator Time()**

to convert a Date object into a Time.

Obviously we can't have two different ways of performing the same conversion. You must choose to define only one of

**Time::Time(Date&)**

**Date::operator Time()**

to convert a Date into a Time object and only one of

**Time::operator int()**

**Date::Date(Time&)**

to convert a Time into a Date. If you define both functions the compiler will object.

## Constructor Casts ...

```
class Time
{
private:
    int hours;
    int minutes;
public:
    Time () : hours(0), minutes(0)
{}
    Time (double);
};

Time::Time (double theTime)
{
    hours = (int) theTime;
    minutes = (int)((theTime -
hours) * 60);
}
```

int

Time

```
int main()
{
    Time t;

    t = (Time) 23.5;
    t = Time (23.5);
}
```

Copyright ©1994-2011 CRS Enterprises

266

When a constructor is used as a cast, the compiler creates a temporary object and uses the constructor to initialize it. Normally constructors are called implicitly, so the cast is easy to spot.

The code fragment

**t = (Time) 23.5;**

uses a cast to create a temporary Time object from the constant 23.5. The temporary object is initialized by the constructor

**Time::Time(double)**

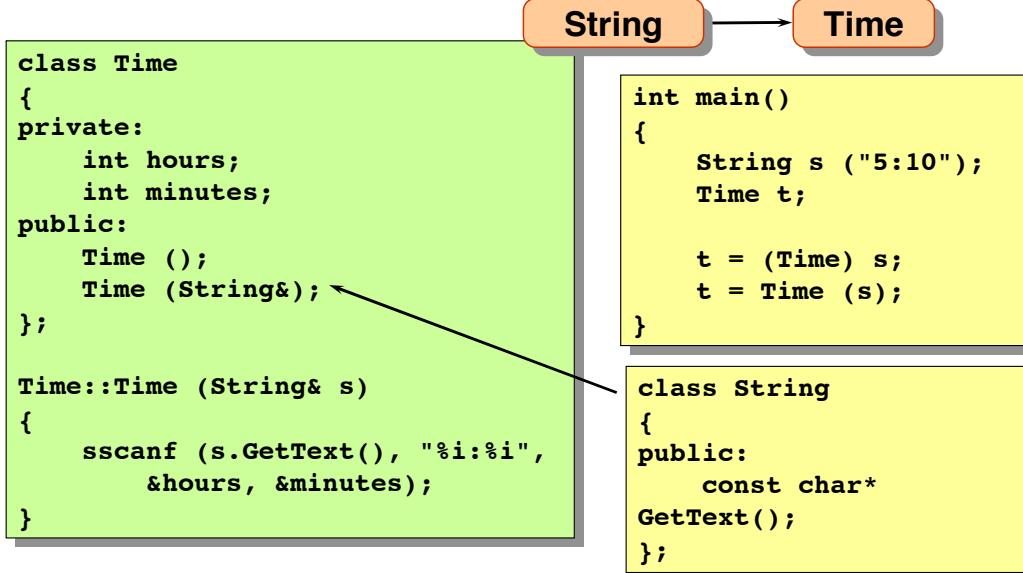
After the assignment to t is completed, the temporary object is destroyed (and hence its destructor called).

Note that the code fragment can be recoded as

**t = Time(23.5)**

if preferred. This is an alternative syntax for the cast.

## ... Constructor Casts



Copyright ©1994-2011 CRS Enterprises

267

When a constructor is used as a cast, the compiler creates a temporary object and uses the constructor to initialize it. Normally constructors are called implicitly, so the cast is easy to spot.

The code fragment

**t = (Time) 23.5;**

uses a cast to create a temporary Time object from the constant 23.5. The temporary object is initialized by the constructor

**Time::Time(double)**

After the assignment to t is completed, the temporary object is destroyed (and hence its destructor called).

Note that the code fragment can be recoded as

**t = Time(23.5)**

if preferred. This is an alternative syntax for the cast.

## Operator Casts ...

```
class Time
{
private:
    int hours;
    int minutes;
public:
    Time (int, int);
    operator double ();
};

Time::operator double ()
{
    return hours +
minutes/60.0;
}
```

**double** ←→ **Time**

```
int main()
{
    double number;
    Time t(3,33);

    number = (double) t;
    number = double (t);
}
```

Copyright ©1994-2011 CRS Enterprises

268

To perform conversions from a class object to a built in type we must use an operator function.

In this example we use the  
**operator double ()**

function to perform the conversion from a Time object to a double. Note that the operator function is called

**operator double**

The double keyword is part of the function name and not part of a parameter list; conversion functions are not permitted to have parameter lists nor return types.

## ... Operator Casts

```
class Time
{
private:
    int hours;
    int minutes;
public:
    Time (int, int);
    operator String ();
};

Time::operator String()
{
    char text[80];
    sprintf(text, "%i:%i",
            hours, minutes);
    String temp(text);
    return temp;
}
```



```
int main()
{
    String s;
    Time t(7,20);

    s =
    (String) t;
    s = String
    (t);
}
```

Casts using operator functions that convert between two classes are less common, but still valid. In this example, we are converting from a Time object to a String object.

When viewed from the Time class, this type of cast is clearly a conversion operation rather than construction and hence

**Time::operator String ()**

is used. From the String class's point of view the operation is construction, so the cast could have been implemented using:

**String::String(Time&) ()**

Obviously, we can only define one of these operator functions or we will create an ambiguous situation for the compiler!

## Overloading [ ]

```

class Array
{
private:
    int* pData;
    int columns;
public:
    Array(int);
    int& operator[ ] (int);
};

Array::Array(int c):
    columns(c),
    pData(new int[c])
{ }

int& operator[ ] (int i)
{
    if (i < 0 || i > columns) ... // error !!
    return pData[i];
}

```

```

int main()
{
    Array a(5);

    a[2] = 3;
    a[4] = 7;

    // what happens
    here ???
    a[100] = 23;
}

```

Copyright ©1994-2011 CRS Enterprises

270

This example shows how to overload the [ ] operator. The Array class encapsulates a C style variable length array and provides a constructor to initialize the Array object.

When overloading the [ ] operator for the class we can easily add bounds checking as shown. If an invalid index is supplied to the function we can either use an assert macro to report the error or throw an exception. Throwing an exception is preferred (see the Exception Handling chapter).

Note that the code in the main program looks like normal C code. However a is really a C++ style array and the code fragment

**a[2] = 3;**

actually calls operator[ ]. Clearly

**a[100] = 23;**

is illegal and will cause operator[ ] to throw an exception.

## Overloading ()

```
class Array
{
private:
    int* pData;
    int rows;
    int columns;
public:
    Array(int, int);
    int& operator() (int, int);
};

Array::Array(int r, int c)
:
    rows(r),
    columns(c),
    pData(new int[r*c])
{};

int& operator() (int i, int j)
{
    return pData[i*columns + j];
}
```

```
int main()
{
    Array v(3,5);

    v(0, 0) = 10;
    v(0, 1) = 20;
    v(1, 0) = v(1, 1);
}
```

Copyright ©1994-2011 CRS Enterprises

271

If we want to simulate multi-dimensional arrays in C++ we would expect to overload the [ ] operator for each dimension. Although this can be done, the resulting code gets very involved. A simpler approach is to overload the () operator.

The trick with overloading the () operator is to treat the multi-dimensional array as a one dimensional array internally. When the array is indexed with [i, j], we simply calculate the offset in the one dimensional array using the formula:

$$\text{offset} = i * \text{columns} + j$$

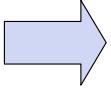
and return a reference to this data at this offset.

## Overloading the ->...

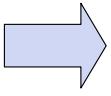
- **Handle-Body idiom**
  - `->` has special treatment
  - overload works in 2 phases

```
Handle h(1, 15);
h->Print();
```

- **Phase 1: unary operator**
  - operates on the *handle*
  - returns a C style pointer *pBody*
  - for use in next phase to access *body*
- **Phase 2: binary operator**
  - code generated by the compiler
  - the C style pointer delegates to the `Print()` method in the *body*



`pBody = handle.operator->()`



`pBody->Print()`

Overloading the `->` operator is an interesting operation in its own right and is essential for understanding the mechanics of managed pointers.

The overload of `->` is based on the handle-body idiom and works in two phases. In the first phase the operator acts solely on the handle object as a unary operator. Although an operator can be programmed to do anything you like, normal practice is to temporarily extract the wrapped raw pointer from the handle and make this the return value

**`pBody = handle.operator->()`.**

In the second phase, the return value is used to delegate to the body, as in

**`pBody->Print()`**

and here the operator is being used as a binary operator. You don't write any code for the second phase; it is generated automatically by the compiler.

## ... Overloading ->

```

class Person {
public:
    void PrintName();
};

class PersonPtr {
private:
    Person* p;
public:
    void operator= (Person* ptr)
        Person* operator-> ();
};

void PersonPtr::operator= (Person* ptr)
{
    p = ptr;
}
Person* PersonPtr::operator-> ()
{
    if (p == NULL) throw "Zero pointer";
    return p;
}

```

```

int main()
{
    Person steve;
    PersonPtr q;

    q = &steve;
    q->PrintName();

    q = NULL;
    q->PrintName();
}

```

Copyright ©1994-2011 CRS Enterprises

273

One of the most frustrating run time errors in C is accidentally dereferencing a NULL pointer and seeing your code crash with a memory access violation. The idea of overloading the `->` operator is to prevent such disasters.

If we wrap up a C pointer in a class and overload the `->` operator we create a smart pointer. Smart pointers check that they are non NULL before allowing themselves to be dereferenced.

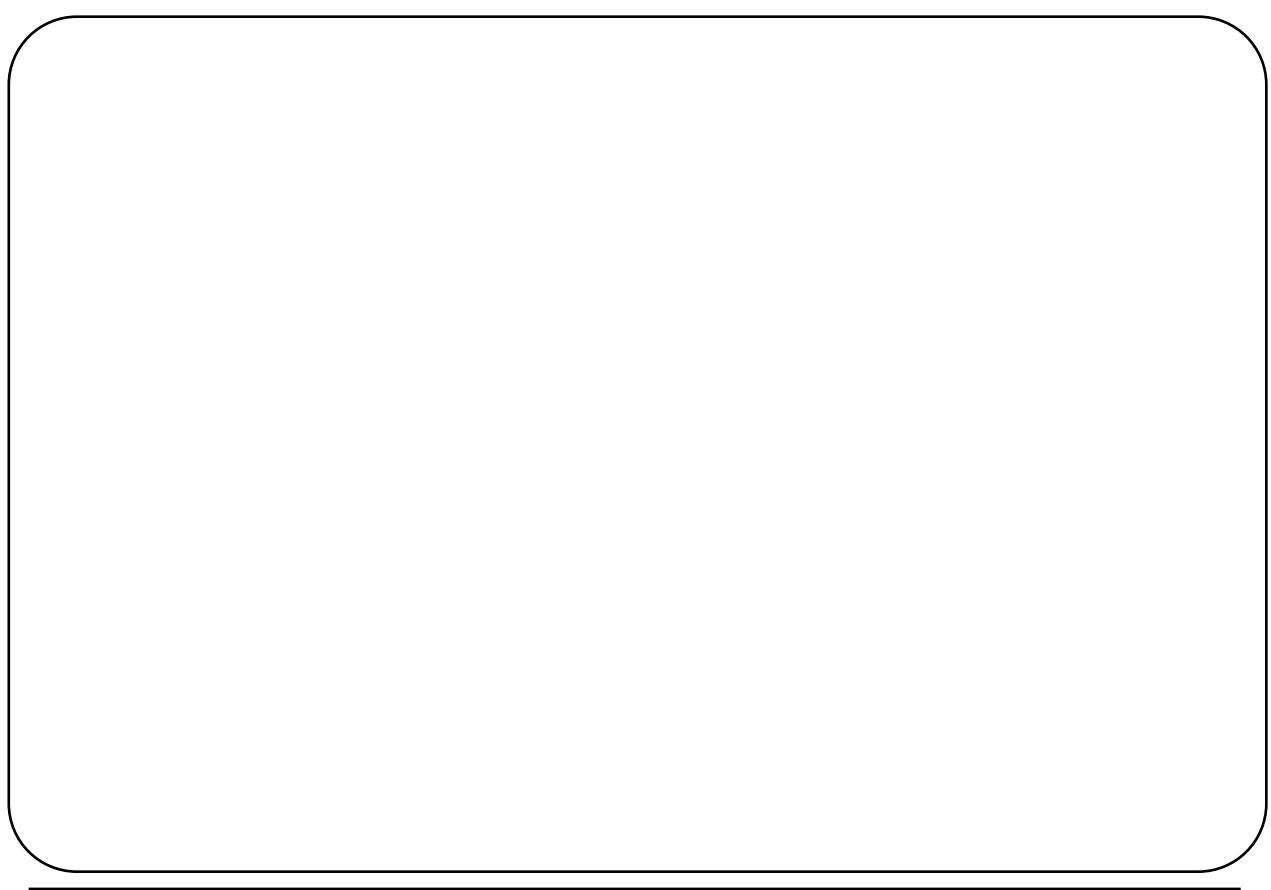
In this example, `q` is a smart pointer. The smart pointer is initialized with a C style pointer to a `Person` object. This pointer is held internally to `q`. When we want to send the `Person` object a message we use the `->` operator. The code fragment

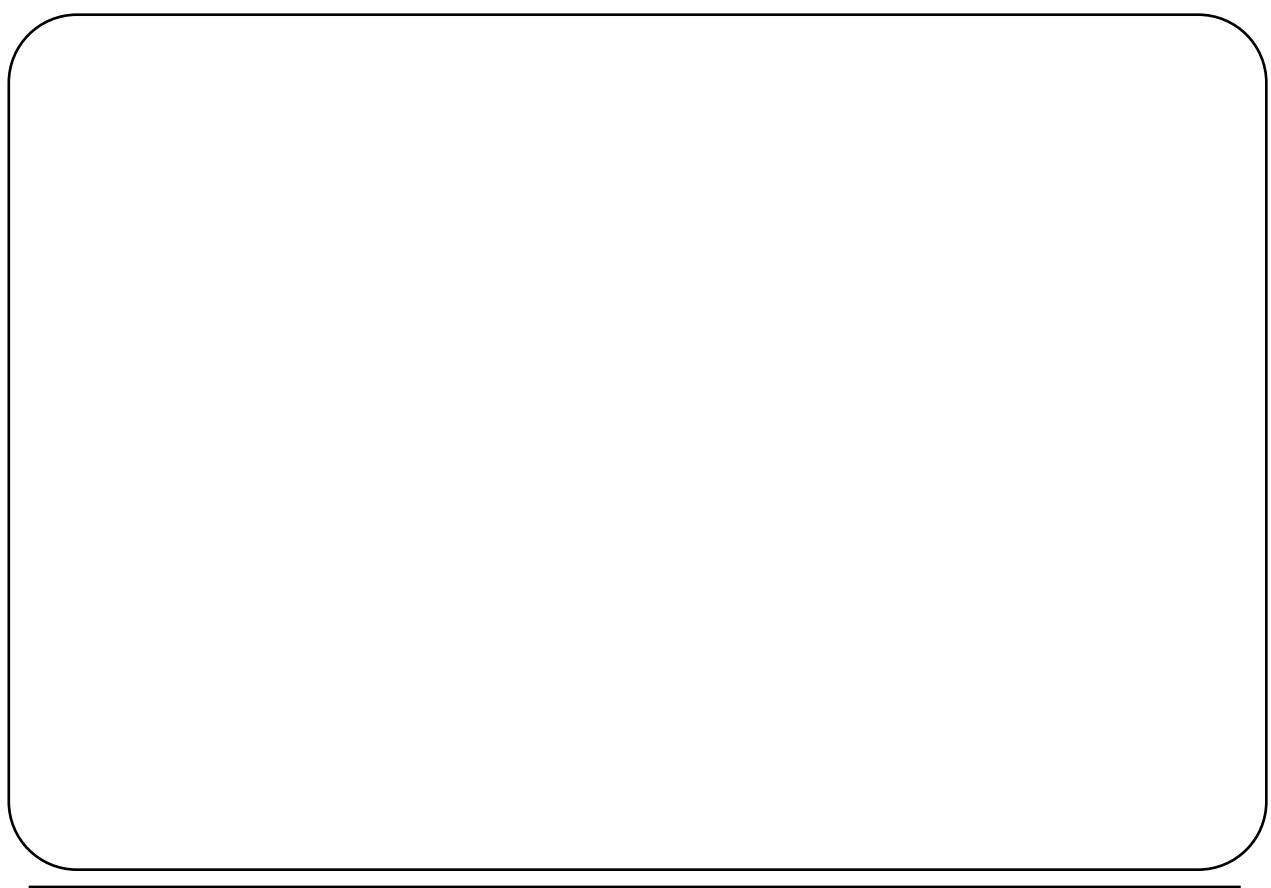
`p->PrintName()`

is interpreted by the compiler as two separate statements:

`Person* ptr = q.operator->()`  
`ptr->PrintName`

The object `q` is sent the unary operator message `operator->()` which returns a C style pointer to the `Person` object. Before this function returns it has an opportunity to check if this pointer is NULL. If it is NULL the function throws an exception, otherwise the function returns normally and the returned C style pointer is used to forward the `PrintName` message to the intended `Person` object.





# 21

## Templates

- **Function Templates**
- **Class Templates**



# 21

Templates were introduced into C++ to simplify the generation of groups of similar functions and classes. Templates do not originate with C++; they have been used in other languages for several years.

Using function templates allows the programmer to construct a group of similar functions by writing a single parameterized function.

Class templates work in a similar way to function templates, except that the programmer must create a template for the class definition (the header information) and every member function. The parameterized class can then be instantiated into many real classes.

## Generic Functions

*Main.cpp*

```
int max (int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
double max (double a, double b)
{
    if (a > b)
        return a;
    else
        return b;
}
Date max (Date a, Date b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Copyright ©1994-2011 CRS Enterprises

278

Consider the three global functions opposite. Each function performs a similar task which is to calculate the maximum of two objects (ints, doubles or Dates). Clearly, the functions only differ in terms of the data types used. The code for each function is identical.

Such functions are ideal candidates for being rewritten as a single parameterized template function.

## Function Templates

### *Max.hpp*

```
template <class TYPE>
TYPE max (TYPE a, TYPE b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

### *Main.cpp*

```
#include "Max.hpp"

int main()
{
    Date d1(10, 4, 1970);
    Date d2(15, 8, 1940);
    cout << max(5, 8)    << endl;
    cout << max(5.1, 8.3)  << endl;
    cout << max(d1, d2)  << endl;
}
```

Copyright ©1994-2011 CRS Enterprises

279

The parameterized function template is written as shown opposite. Note the use of the template keyword to introduce the function. The template parameters are enclosed in angled braces (to distinguish them from function parameters).

Templates are instantiated simply by calling the real function. The call

**max(5, 8)**

will instantiate the template for ints, whereas

**max(5.1, 8.3)**

**max(d1, d2)**

will instantiate the template for doubles and Dates respectively.

Note that TYPE is a user defined name for the template parameters and not a C++ keyword.

## Using References

*Main.cpp*

```
template <class T>
const T& max (const T& a, const T& b)
{
    if (a > b)
        return a;
    else
        return b;
}

int main()
{
    Date d1(10, 4, 1970);
    Date d2(15, 8, 1940);
    cout << max(5, 8) << endl;
    cout << max(5.1, 8.3) << endl;
    cout << max(d1, d2) << endl;
}
```

Copyright ©1994-2011 CRS Enterprises

280

The previous template works well for built in types, but can prove inefficient for user defined types. This is because the template passes copies of the two objects to Max.

It is much better if we pass objects by reference rather than value (this has little effect on built in types). The new template is shown opposite.

## Mixed Parameters

### *Max.hpp*

```
template <class T1, class T2>
const T1& max (const T1& a, const T2& b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

### *Main.cpp*

```
int main()
{
    Date d(10, 4, 1970);
    Time t(340071);

    cout << max(d, t) << endl;
}
```

Templates are not restricted to working with a single parameter. Consider the template opposite. This template uses two types, T1 and T2. This allows the template to find the maximum of two objects that may belong to different classes.

Of course, this only makes sense if the `>` operator is properly defined to allow a comparison between the objects. In this example the template instantiation

`max(d, t)`

finds the maximum of a Date and a Time object. The `>` operator must be overloaded, probably as a friend function of both the Date and Time classes:

```
class Time
{
friend Date& operator<(Date&, Time&);

};

class Date
{
friend Date& operator<(Date&, Time&);

};
```

Note that you must choose to return a Date& or a Time& object. This friend function returns a Date&. The operator`<<` function must also be overloaded for

`cout << max(d, t)`

## Generic Classes ...

```

class ArrayOfInts
{
private:
    int array[3];
public:
    ArrayOfInts(int);
    void Print();
};

class ArrayOfDoubles
{
private:
    double array[3];
public:
    ArrayOfDoubles(double);
    void Print();
};

```

```

int main()
{
    ArrayOfInts a(3);
    ArrayOfDoubles b(5.5);
}

```

Copyright ©1994-2011 CRS Enterprises

282

Consider the two classes `ArrayOfInts` and `ArrayOfDoubles` shown opposite. These classes have a great deal in common and are suitable candidates for being converted into a class template.

However, with class templates we have more work to accomplish than with function templates. Not only do we need to parameterize the class definitions opposite, but we will need to write templates for all common methods, including constructors.

## ... Generic Classes

```

ArrayOfInts::ArrayOfInts(int x0) {
    for (int i = 0; i < 3; i++)
        array[i] = x0;
}
void ArrayOfInts::Print() {
    for (int i = 0; i < 3; i++)
        cout << array[i] << '\t'
    cout << endl;
}

ArrayOfDoubles::ArrayOfDoubles(double x0) {
    for (int i = 0; i < 3; i++)
        array[i] = x0;
}
void ArrayOfDoubles::Print() {
    for (int i = 0; i < 3; i++)
        cout << array[i] << '\t'
    cout << endl;
}

```

Copyright ©1994-2011 CRS Enterprises

283

Here, we see the methods of each class before parameterization. The two constructors differ in their names and in their parameter lists

**IntArrayOfInts::IntArrayOfInts(int, int, int)**  
**ArrayOfDoubles::ArrayOfDoubles(double, double, double)**

but the Print functions only differ in their class name

**IntArrayOfInts::Print()**  
**ArrayOfDoubles::Print()**

## Class Templates

```

template <class T> class Array
{
private:
    T array[3];
public:
    Array(T);
    void Print();
};

template <class T> Array<T>::Array(T x0)
{
    for (int i = 0; i < 3; i++)
        array[i] = x0;
}

template <class T> void Array<T>::Print()
{
    for (int i = 0; i < 3; i++)
        cout << array[i] << '\t';
    cout << endl;
}

```

Copyright ©1994-2011 CRS Enterprises

284

The class definitions and the two methods are parameterized as shown opposite. Instantiating the template with TYPE equal to int will generate:

```

class Array
{
private:
    int array[3];
public:
    Array(int);
    void Print();
};

Array<int>::Array(int x0)
{
    for (int i = 0; i < 3; i++)
        array[i] = x0;
}

void Array<int>::Print()
{
    for (int i = 0; i < 3; i++)
        cout << array[i] << '\t';
    cout << endl;
}

```

Note that the class appears to have two names. Is the class called Array or Array<int> ? In fact the class is known as

**Array<int>**

## Other Parameter Types

```

template <class T, int SIZE> class Array
{
private:
    T array[SIZE];
public:
    Array(T);
    void Print();
};

template <class T, int SIZE> Array<T , SIZE>::Array(T x0)
{
    for (int i = 0; i < SIZE; i++)
        array[i] = x0;
}

template <class T, int SIZE> void Array<T , SIZE >::Print()
{
    for (int i = 0; i < SIZE; i++)
        cout << array[i] << '\t';
    cout << endl;
}

```

Copyright ©1994-2011 CRS Enterprises

285

Parameters to templates are not restricted to class names. In the example opposite, the size of the array is parameterized in the template

**template <class T, int SIZE>**

This means we can create classes from the template that encapsulate an array of arbitrary class type T and these arrays can have any size. You can see why templates are so popular - with minimal effort we have created an extremely powerful set of classes!

Note that the class name is becoming a little complicated. The constructor

**template <class T, int SIZE>**  
**Array<T , SIZE>::Array(T x0)**

refers to the class

**Array<T , SIZE>**

## Instantiating Templates

*Main.cpp*

```
int main()
{
    Date newYear(1, 1, 2000);

    Array<int, 3>      a(8);
    Array<double, 5>   b(8.888);
    Array<Date, 4>     c(newYear);

    a.Print();
    b.Print();
    c.Print();
}
```

Copyright ©1994-2011 CRS Enterprises

286

The class template is instantiated as shown opposite. The line

**Array<int,3> a(8);**

creates the class **Array<int,3>**, together with its class definition, its constructor and its Print member function. Note that the constructor has a slightly different name from that of the class. The constructor is called **Array** and not **Array<int,3>** as we would have expected.

The line

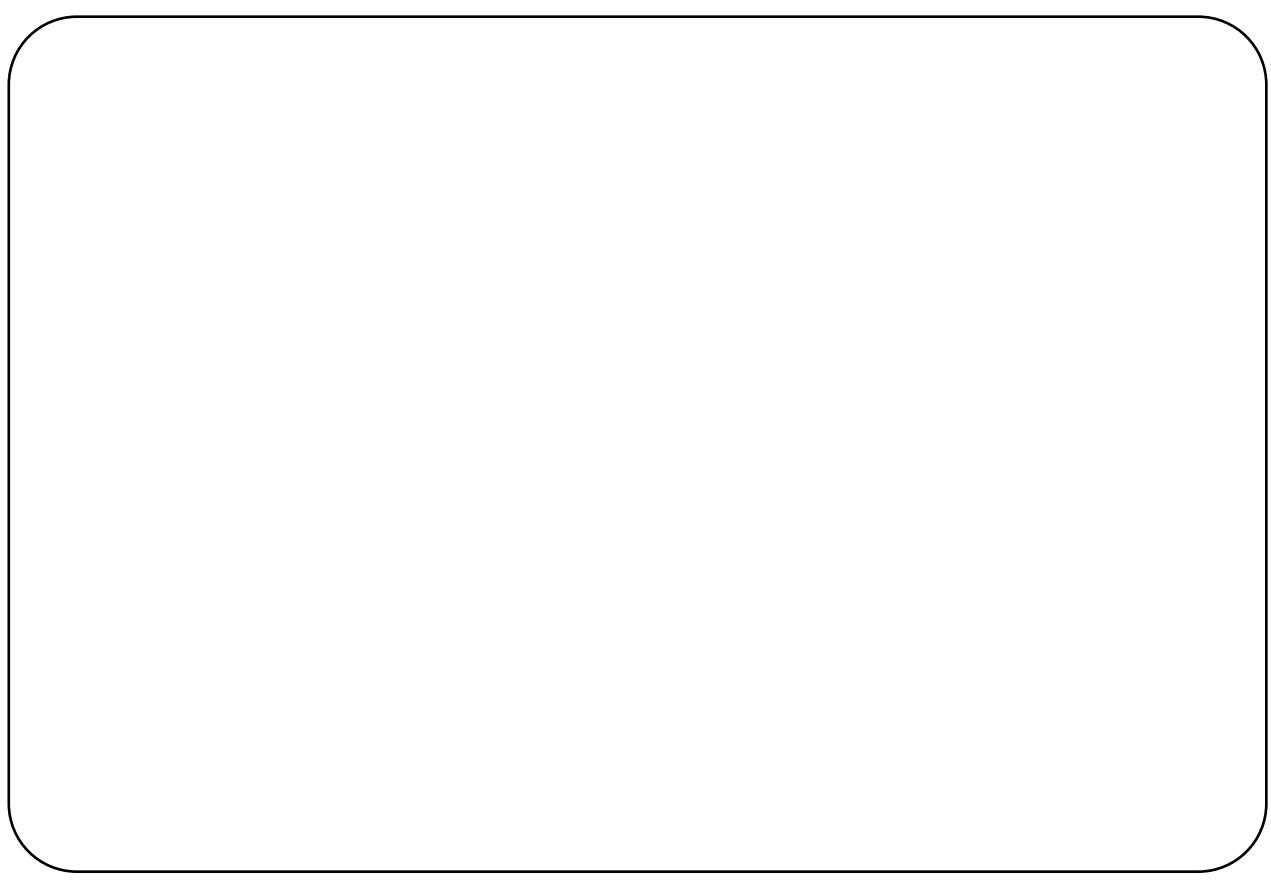
**Array<double,5> b(8.888);**

instantiates the class **Array<double,5>**, together with its class definition, its constructor and its Print member function.

Similarly

**Array<Date,4> c(newYear);**

creates an object **c** of class **Array<Date,4>**. Each element of the **c** is initialized to the **newYear** object using the classes copy constructor.



# 22

## Exception Handling

- **Defining Exceptions**
  - Standard Exception Hierarchy
  
- **Grouping Exception Types**
  - in classes
  - in namespaces
  - in inheritance hierarchies
  
- **Special Cases**
  - heap based objects
  - exceptions in constructors
  - uncaught exceptions
  
- **Throw Lists**



**22**

Copyright ©1994-2011 CRS Enterprises

289

Error handling has always been a difficult topic in software engineering. In the C programming language, the programmer is responsible for checking return codes from functions to detect errors. This approach is both inflexible and unwieldy and to make matters worse software engineers do not check return codes rigorously.

C++ takes an entirely different approach to error handling: the method of exception handling. This approach is flexible and efficient and is based on similar methods that have been developed in other programming languages such as ADA.

C++ uses the keywords try, throw and catch in exception handling. The try block defines the section of source code checked for errors (exceptions). The throw statement is used to generate an exception (an error object gets thrown after an exception). A catch block is an exception handler.

## Exceptions

- **Separate error detection from error correction**
  - detection often made in library code
  - library author can detect error, but doesn't know how to correct
  - correction made in user code
  - user can't detect error, but does know how to correct error
- **try**
  - to mark a block as possible source of error
  - try blocks can be nested
- **throw**
  - when error is detected
- **catch**
  - to provide recovery

All too often, in traditional C programming, normal code is cluttered with error handling code and the two cannot easily be decoupled. The principal idea behind C++'s exception handling capability is to provide separation of error detection from error correction.

Errors can occur anywhere in a program, but typically errors occur in deeply nested routines such as library modules. When these errors occur, they are usually obvious to the code author, but error recovery is a different matter. Library authors can detect errors, but do not know what the caller wants to do to recover from the error. The caller on the other hand is in the opposite position. The caller finds it difficult to detect errors, but knows what to do if one occurs.

With exception handling separation of responsibilities and decoupling of error detection and correction is easily achieved.

try blocks are used to mark a block of code as a possible source of error. Conceptually, the try block includes all nested method calls.

throw is used when an error is detected.

catch is used to define code used for error recovery.

## Simple Example

```

int main()
{
    Array a;
    int index;
    while(true)
    {
        try
        {
            cout << "Enter array index: ";
            cin >> index;
            cout << a[index] << endl;
        }
        catch (char* errorMessage)
        {
            cout << errorMessage << endl;
        }
    }
}

```

```

class Array
{
private:
    int array[10];
public:
    Array();
    int operator[ ] (int);
};

int Array::operator[ ] (int index)
{
    if (index < 0) throw "index underflow";
    if (index >= 10) throw "index overflow";

    return array[index];
}
...

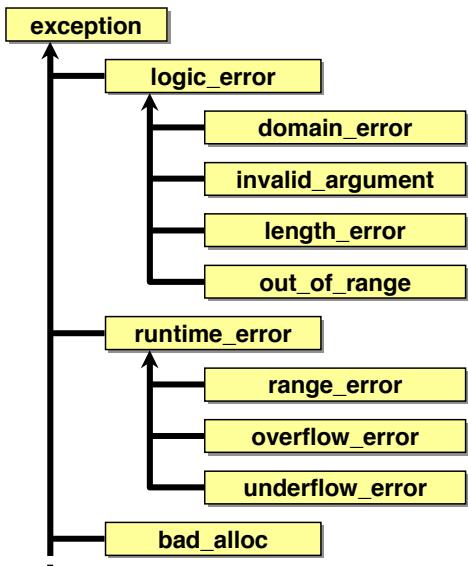
```

Copyright ©1994-2011 CRS Enterprises

291

In this simple example an error could potentially occur anywhere in the try block, but in practice the error will occur when the index for the array wrapper object goes out of bounds. The library author (of class array) can detect the array bounds error and throw an exception. The exception is caught in the main program.

## The Standard Exception Hierarchy



- **recommendation**
  - use the standard exception classes
  - or inherit from these classes
  - don't write your own

Although you can write your own exception classes it is recommended that you use (or inherit from) the standard exception classes from the standard library:

`exception` is the base class for exceptions

`runtime_error` is the base class for general errors in program execution

`logic_error` is the base class for exceptions that represent a fault in the program's logic, such as a violated function precondition

`runtime_error` is the base class for general errors in program execution

`bad_alloc` is the exception thrown when an attempt to allocate an object using `new` fails

## Grouping Exception Types

- **Part of a namespace**
  - exceptions common to a library can be defined within its namespace
- **Part of a class**
  - nest within the class
- **Relating by inheritance**
  - define a hierarchy of exception types

```
namespace utilities
{
    class stack
    {
        public:
            struct error {};
            struct empty : error {};
            void pop();
            //...
    };
}
```

```
class Error { ... }
class MathError : public Error
{
public:
    virtual void Diagnose() {}
};
```

It is not recommended to throw primitive values, such as strings and integers; class based objects can carry more information about the exception and do not rely on the use of error codes.

If an exception is used exclusively by a class, the exception type should be nested within the scope of this class. Similarly if an exception is used frequently throughout a library then it should be made part of the library's namespace.

Furthermore, exceptions may also be grouped using inheritance, with specialisations (e.g. MathsError or RangeError from Error) deriving from a common base class.

## Exceptions Using Inheritance

```
int main()
{
    try
    {
        // code that throws MathError
        catch(Error& e)
        {
            e.Diagnose();
        }
    }
```

- relate exceptions
  - using inheritance
- use polymorphism
  - to distinguish exceptions
  - e.g. **Diagnose()** method

```
class Error
{
public:
    virtual void Diagnose() { ... }
};

class MathError : public Error
{
public:
    virtual void Diagnose() { ... }
};

class FileError : public Error
{
public:
    virtual void Diagnose() { ... }
};
```

When you define an exception hierarchy it is normal practice to write catch handlers for each of the derived classes. However, with a large hierarchy this will give rise to an untidy list of catch handlers at the end of each try block. An alternative, well worth considering, is to provide a catch handler for the base class and use polymorphism to direct the error to the correct derived class.

In the above example, although we have only provided a catch handler for Error, if a MathError is thrown, then the polymorphic call

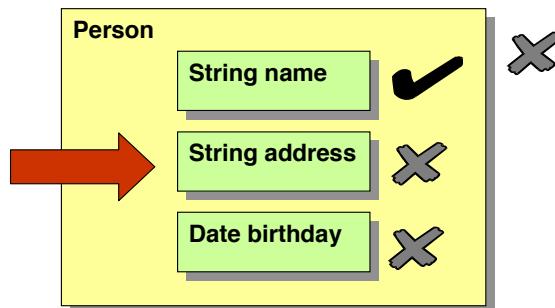
**e.Diagnose()**

will direct error handling to the MathError class.

## Exceptions in Constructors

- If a constructor throws an exception
  - object is marked as non initialised
  - destructor not called
  - memory reclaimed
- Composite objects clean up fully initialised components
  - destructor is called
  - memory reclaimed
  - other components follow rule above

*exception thrown  
in constructor*



Copyright ©1994-2011 CRS Enterprises

295

If an exception is thrown inside the constructor of a stack based object, the object is marked as non initialised. In such cases it doesn't make sense for the object's destructor to be called, but its memory is automatically reclaimed by the compiler.

Composite objects are more complicated. If an exception is thrown inside the constructor of a component object then the above rule applies to the component, the composite object itself and any components not yet initialised. Any components already initialised will be marked as initialised and will be cleaned up (destructor called and memory reclaimed) when the component object goes out of scope at the end of the catch handler.

## Problems with Heap Based Objects

- **stack based objects**
  - cleaned up automatically
- **heap based objects**
  - do not get cleaned up automatically
  - alternatively use smart pointers

```
void f2()
{
    Person hilda("Hilda");
    throw "some kind of problem";
}

void f1()
{
    Person* pSteve = new Person("Steve");
    f2();
    delete pSteve;
}
```

*not cleaned up automatically*



Copyright ©1994-2011 CRS Enterprises

296

The compiler ensures that all stack based objects are cleaned up automatically. However, the compiler is powerless to clean up objects created with the new operator at run time. Hence heap based objects do not get cleaned up automatically.

Cleaning up heap based objects is problematical; it appears that hand crafted code is required. However a generalised method of cleaning up heap based objects can be formulated using smart pointers. This will be addressed elsewhere in the course.

## Uncaught Exceptions

- **uncaught exceptions**
  - propagate out of *main*...
- **the *terminate* function is called**
  - calls *abort* by default
  - use *set\_terminate*
  - to register an alternative callback
- **use catch all**
  - *catch(...)*
  - must be the last handler in the block

```
int main()
{
    set_terminate(myHandler);
    ...
}
```

```
int main()
{
    try
    {
        ...
    }
    catch(...)
    {
        cerr << "uncaught" << endl;
    }
    ...
}
```

If an exception is thrown and there is no associated handler, the exception propagates out of the main program and the C++ standard library function *terminate* will be called. This in turn calls the *abort* function and the program will end abnormally.

You can customise this behaviour by using *set\_terminate* to register an alternative handler to be called in the event of an unhandled exception.

Alternatively, you can use the catch all syntax

**catch(...)**

to catch an unhandled exception. This catch clause must be the last handler for the try block.

## Throw Lists

- Specify which exceptions are thrown

```
class stack
{
public:
    ...
    void pop() throw(empty, bad_alloc);           may throw empty or bad_alloc
    void clear();                                may throw anything
    size_t size() const throw();                  throws nothing
    //...
};
```

- Exceptions are checked against the *throw* list at runtime
  - exceptions not listed generate a call to *unexpected*
  - unexpected* calls *terminate*
  - unless alternative callback is registered using *set\_unexpected*

You can specify the list of exceptions that may be thrown on a method invocation. The throw specification is part of a function's enforceable contract.

The throw list for stack::pop states that it will only generate empty or bad\_alloc exceptions, or exceptions derived from these types.

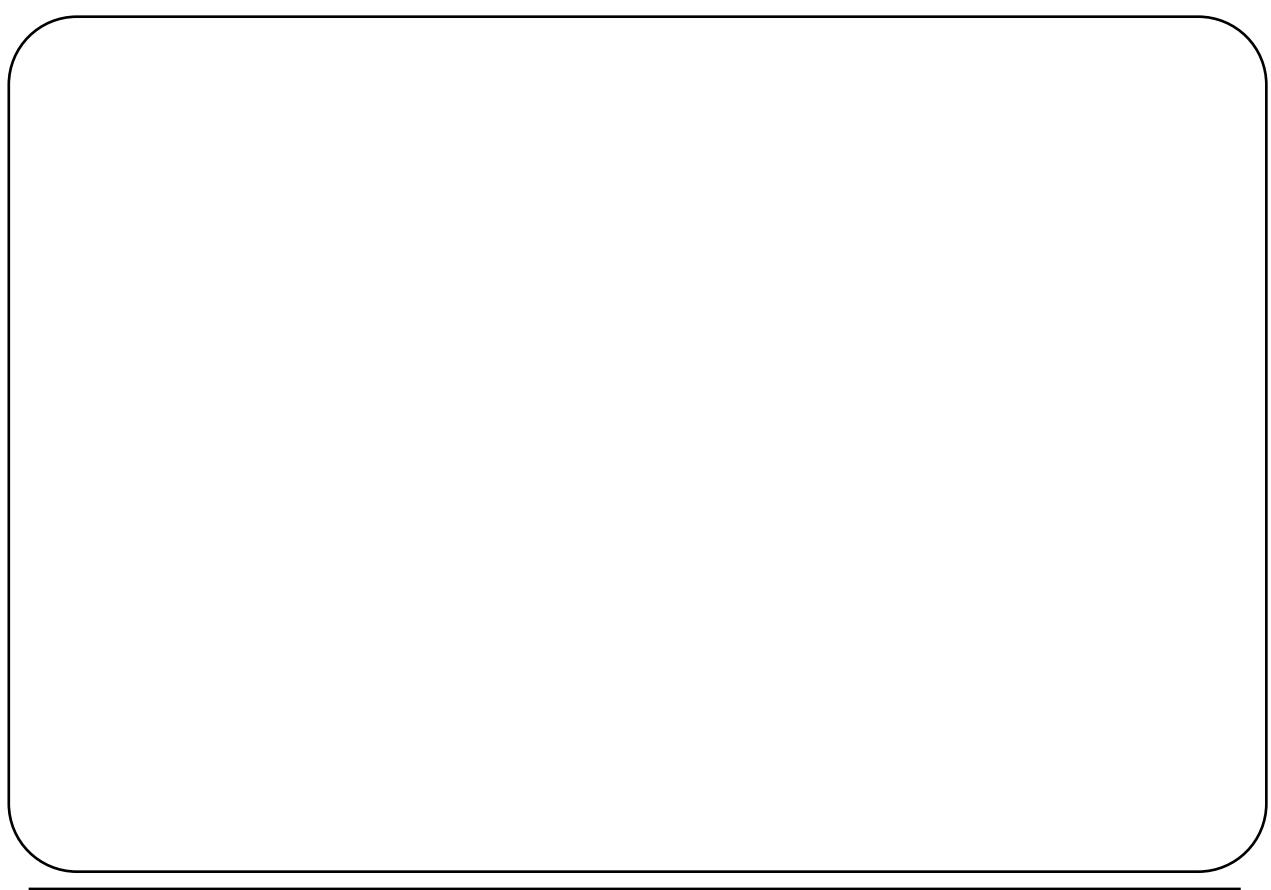
The absence of a throw list, as in stack::clear implies that any exception may be thrown.

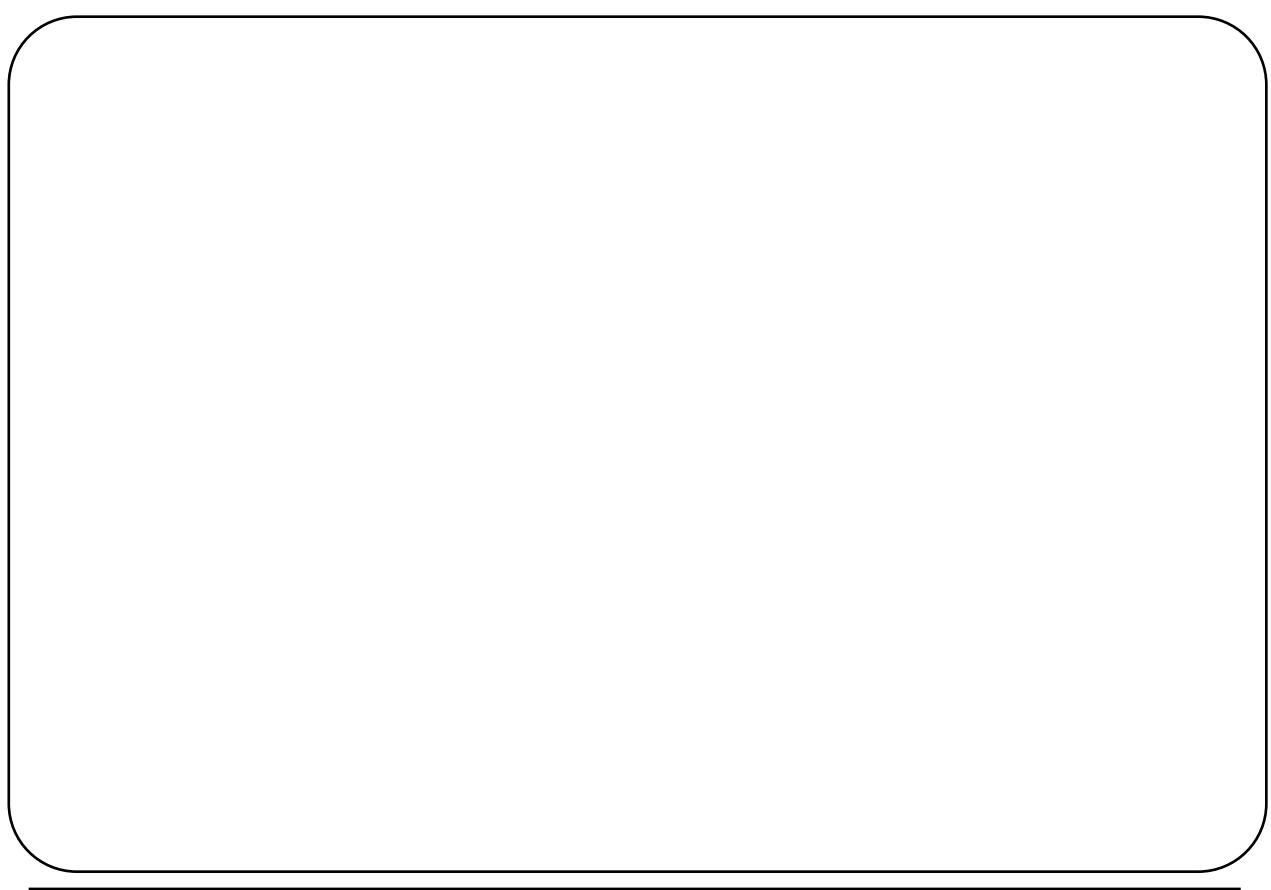
To indicate that no exceptions are thrown an empty list must be used, as with stack::size.

The throw list is not enforced by the compiler; transgressions are checked and handled at runtime. If a method throws any exception not listed in its exception specification, the standard C++ function *unexpected* is called. The default behaviour for *unexpected* is to call *terminate*. Alternatively, you may register a callback using

**set\_unexpected(myHandler)**

It should be mentioned that many programmers do not approve of the throw list mechanism, mainly because throw lists are not enforced by the compiler.





# Appendix

# A1

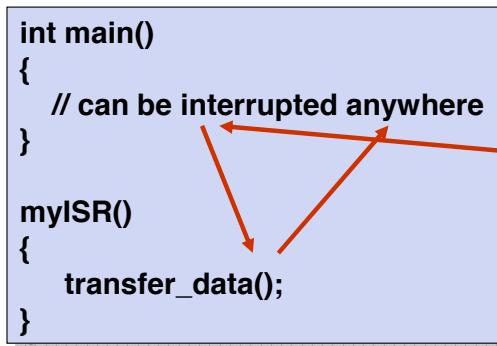
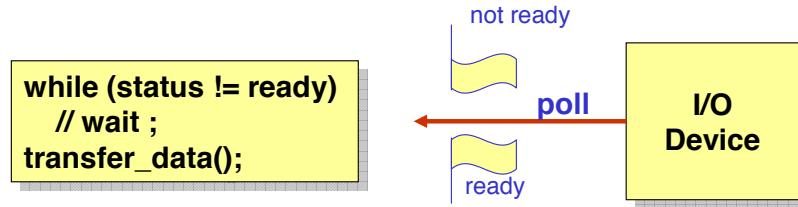
## Interrupt Handling

- **Interrupts or Polling**
- **Interrupt Models**
- **Vector Interrupt Controllers**
- **Writing ISRs**

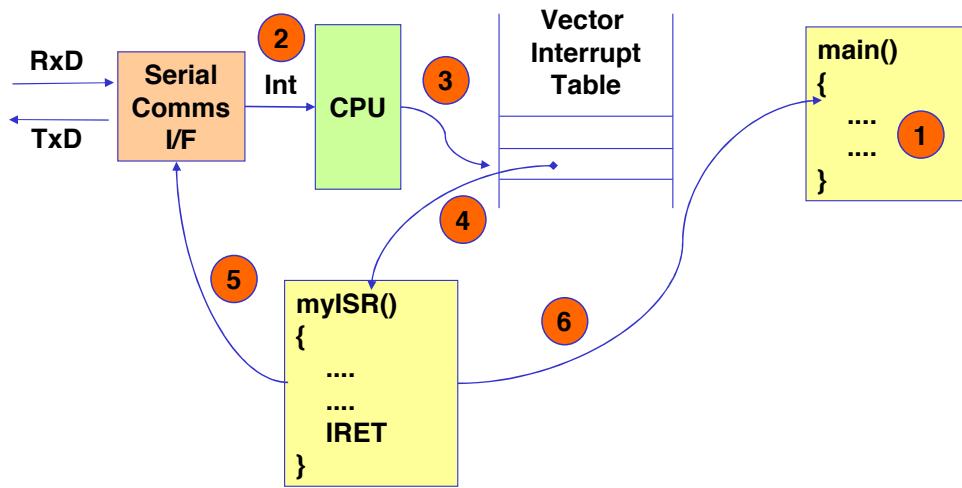


**A1**

## Interrupts and Polling

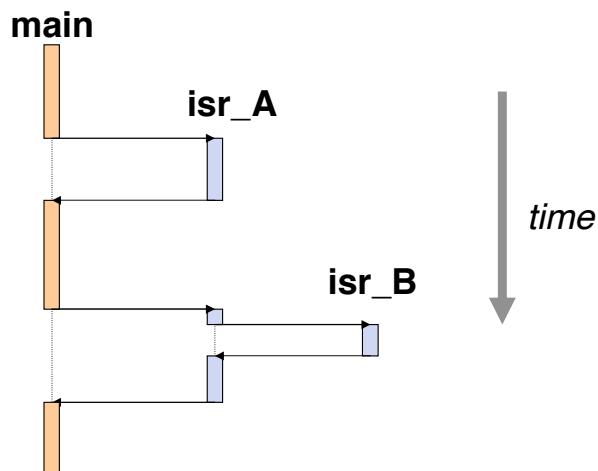


## Interrupt Sequence



## Nested Interrupts

- Nested Interrupts are common
  - many processor support nesting of interrupts
  - interrupt may occur while we are in an ISR

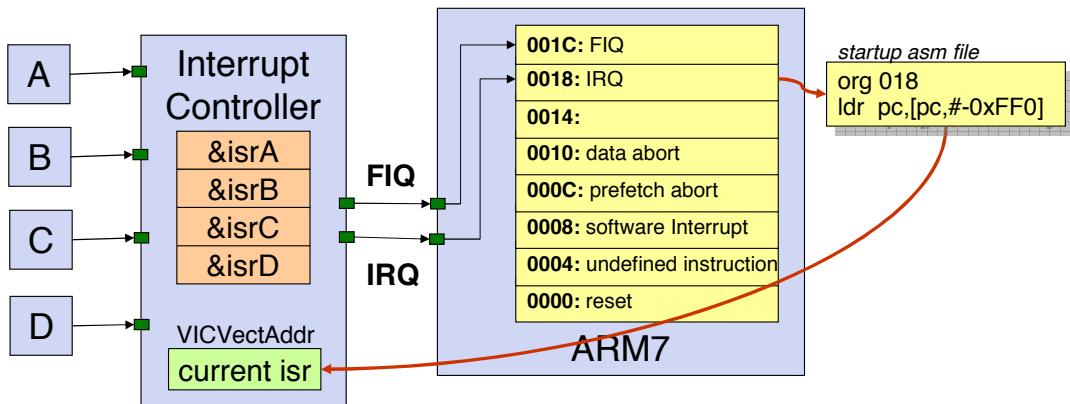


Copyright ©1994-2011 CRS Enterprises

306

## Vectored Interrupt Controllers (VIC)

- **Interrupt controllers are independent processor**
- **Each VIC may be different**
  - no standard defined



Copyright ©1994-2011 CRS Enterprises

307

The vector table might be a bare list of ISR addresses, or it may contain jumps to those addresses, depending upon the processor. In some architectures, the vectors (or some of them) are held within the interrupt controller rather than in the processor's normal memory space. A mixture of these two approaches is common. For example, ARM reserves the lowest part of memory for vectors for different classes of exceptions (including interrupts), but a separate controller (VIC) holds those relating to particular interrupt sources.

The one-instruction jump to the ISR requires some sleight of hand. For example on the LP21xx family, all support a Vectored Interrupt Controller (VIC) external to the core (but on the same silicon). The VIC has a register that makes available the current ISR address. This is located at 0xFFFF030.

The instruction: `ldr pc, [pc,#-0xFF0]`

loads the program counter (pc) register from memory with the value stored at the current pc – 0xFF0. This is  $0x18 - 0xFF0 = 0xFFFF028$  (not 0xFFFF030). However as the ARM7 has the 3-stage pipeline, the current pc is actually 0x20 ( $0x20 - 0xFF0 = 0xFFFF030$ ).

## Interrupt Routines

- Many variations
  - each compiler has its own format

```

/* GNU GCC */
void myISR( ) __attribute__ ((interrupt));

void myISR()
{
    //ISR code
}

/* Renesas */
#pragma interrupt (myISR)
void myISR()
{
    //ISR code
}

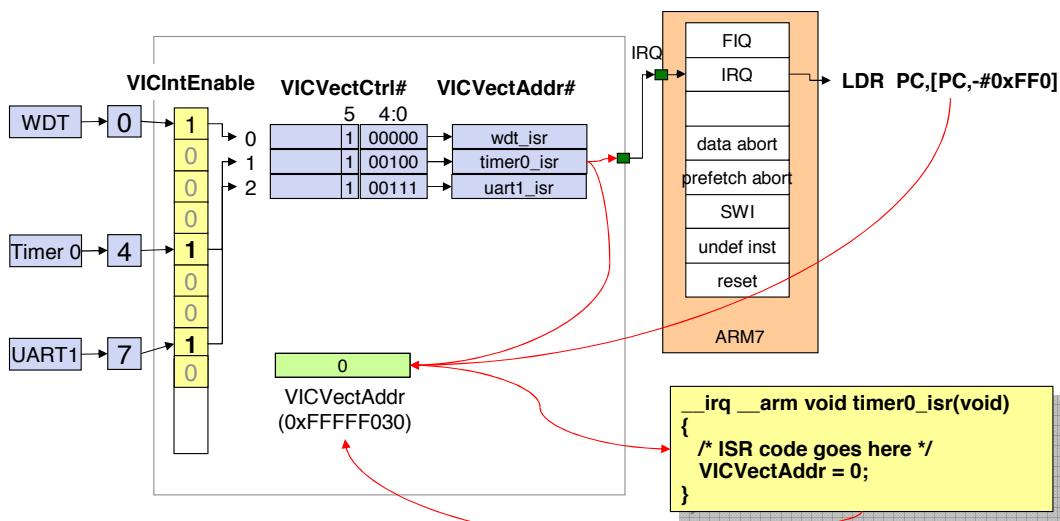
/* ARM ADS */
__irq void myISR( )
{
    //ISR code
}

```

Each compiler has its own way of tagging a function as an ISR.

The 'correct' approach is through the use of pragmas (compiler directives).

## VIC Interrupt Sequence



Copyright ©1994-2011 CRS Enterprises

309

## Writing the ISR

```
Timer::Timer()
{
    pRegs->TCR = 0; // Timer off
    pRegs->MR0 = 45000000; // 3 sec count
    pRegs->MCR = 3;

    __disable_interrupt();
    VICVectAddr0 = (unsigned long) Timer::isr;
    VICVectCtl0 = 0x24; // Map channel 4 (Timer 0) to vectored
                        // IRQ slot 0 and enable slot
    VICIntEnable = 0x10; // Enable channel 4
    VICVectAddr = 0;
    __enable_interrupt();

    pRegs->TCR = 1; // Timer on
};
```

```
class Timer
{
public:
    Timer (unsigned long address, Motor& motor);
    __irq __arm static void isr( );
    ...
}
```

```
__irq __arm void Timer::isr ()
{
    stp->myMotor.ChangeDirection ();
    stp->pRegs->IR = 1; // Clear timer interrupt flag
    VICVectAddr = 0; // Acknowledge interrupt
}
```

## General Considerations

- **ISRs must be quick**
  - hardware needs to be serviced
  - avoid performing unnecessary work
    - schedule work to be performed outside ISR
    - set flags rather than transferring data
- **ISRs may have to mask other interrupts**
  - to avoid recursion and simplify logic
  - more reason to make ISR run quickly
- **ISRs potentially can run at any time**
  - must be careful to synchronize data access
  - essentially equivalent to execution in a parallel thread





# A2

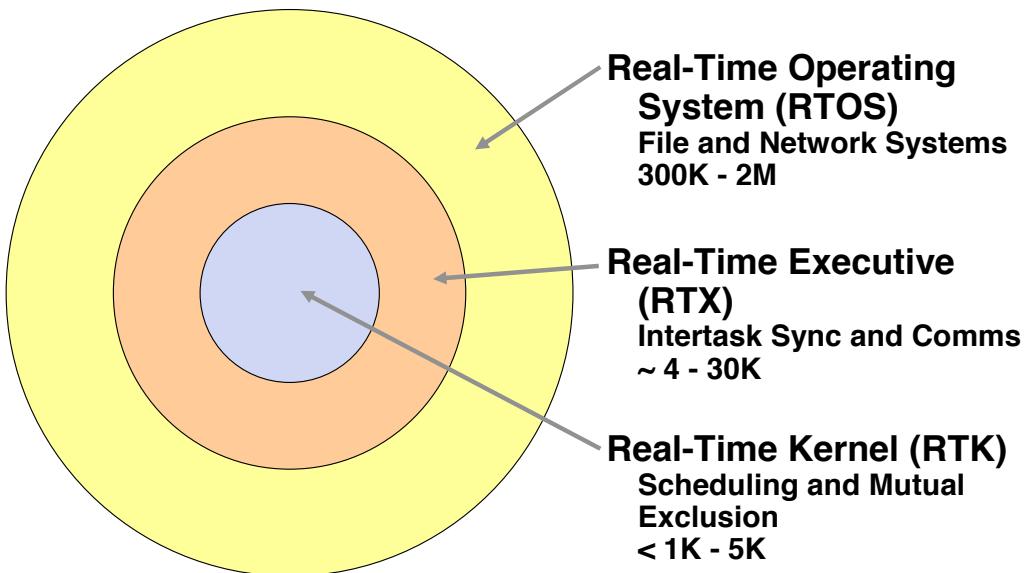
## Real Time Operating Systems

- Key Components
- Scheduler
- Process map
- Paging
- Multi Core Architectures
- Memory Mapped Devices



**A2**

## Structure of an RTOS



## RTOS Scheduler

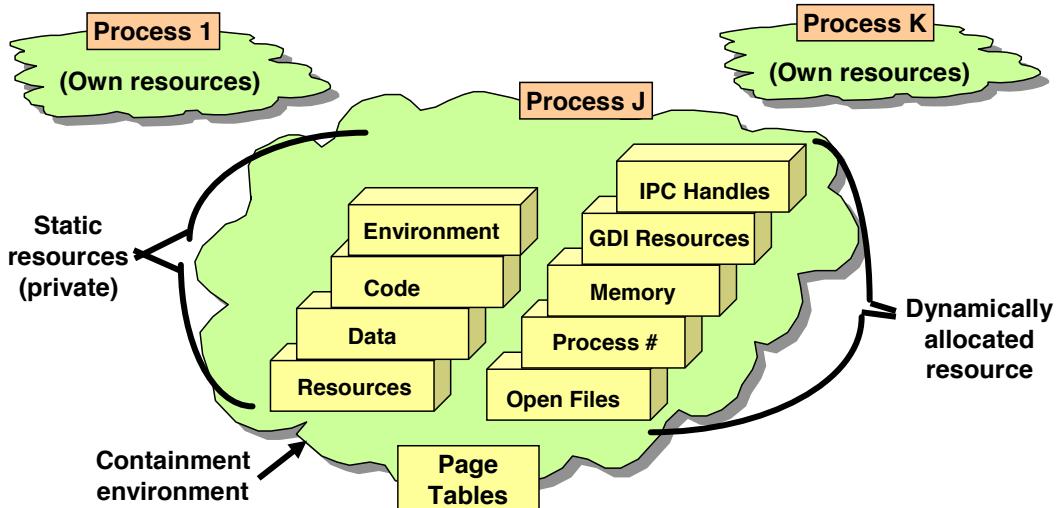
- **Primary scheduler**
  - Gives out *time quanta*s of CPU to competing threads based on thread priority
  - Pre-emption occurs if thread becomes blocked, or higher priority thread becomes available
  - Round robin for threads of same priority
- **Time-Slice Scheduler**
  - Advantage to threads in the foreground process
  - Boosts threads to prevent starvation

Threads are given CPU time by the time-slice scheduler according to their priority, scheduling higher-priority threads before those of lower priority. The scheduler gives out ‘time quanta’ of CPU time to competing threads based on their priorities. The scheduler decides which thread receives the next CPU time quantum, at the end of each time quantum, or if the currently-executing thread becomes blocked, or if a higher-priority thread becomes runnable. It always gives control to the highest priority runnable thread. If several runnable threads have the same priority, the one that has been waiting longest gets the time slice, i.e ‘round-robin’. A runnable thread is one that is not waiting, e.g. blocked on input, or waiting for a semaphore to clear. Non-runnable threads are not considered for CPU time until they become runnable.

Windows 95 deploys time quanta of approximately 20ms depending on the scheduling algorithm. Windows NT Workstation (4.x) uses longer timeslices for foreground applications than background applications. The quantum spread is 3x, 2x or 1x (disabled) depending on the system control panel applet. Windows NT server uses 6x the quantum size of Workstation or 2x that of a Workstation foreground timeslice. Idle priority class applications (screen savers etc) always get the 1x quantum. Aside from the system applet, the thread boosting for foreground applications can be disabled by using the API call SetThreadPriorityBoost() or SetProcessPriorityBoost().

## A Process

- A resource-containment environment



Copyright ©1994-2011 CRS Enterprises

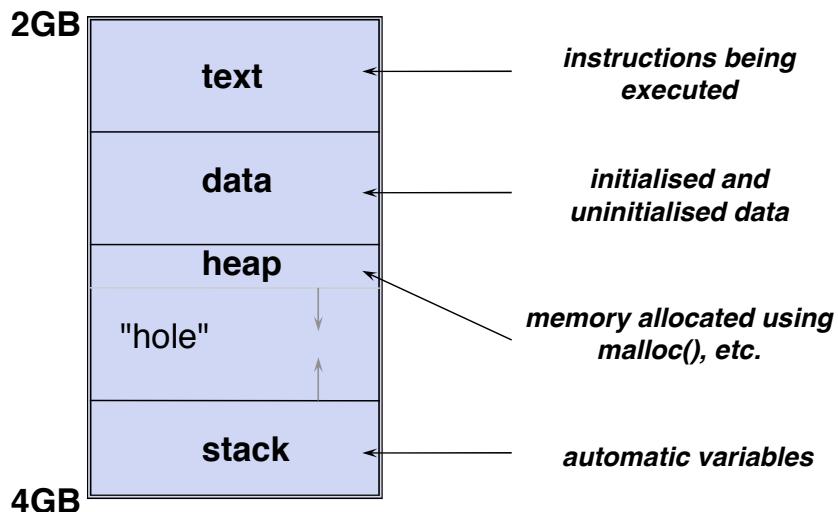
318

A process is a ‘resource ownership and containment environment’ that defines a program that is loaded into memory and is executing. A process defines the context in which the execution occurs, and owns statically-allocated resources, such as code, data and stack space, or dynamically allocated resources, such as files, memory and pipes.

Windows NT maintains a unique set of page tables for each process that define a private 4GB logical, or virtual, address space and map it on to available physical memory. A process can only address memory that is referenced by an entry in its page table, so virtual memory owned by one process is invisible to all other processes, ensuring that processes are protected from each other.

The work of a process is done on its behalf by its threads of execution. Each process has at least one thread of execution. When a process is started up it has a single thread called the ‘primary’ thread, but additional threads can be created. Because of policies imposed by the C run-time libraries and Win32, this primary thread has some attributes which differentiate it from subsequently created threads; it is, for instance, normally responsible for ending the process. However, Windows NT does not consider the primary thread as anything special, and will only terminate the process when all threads have terminated. There is more on terminating processes and threads later.

## Memory Map of a Process



Copyright ©1994-2011 CRS Enterprises

319

A Unix process executes in its own virtual memory or virtual address space. The virtual memory is a linear address space, (i.e. addressed from 0 upwards). Its size will depend on the implementation, but modern systems allow processes to use 32 bits to specify a virtual address, giving a virtual address space of some 4GB (the first 2GB is usually reserved by the kernel).

The virtual memory of a process is organised into sections or "segments", as shown above. This layout, and the addressing scheme to achieve it, is constructed by the link editor (ld), and the correct memory mappings are established when a program is executed (more about this later).

The text portion of a process is the area of memory containing the instructions being executed. The virtual-memory system arranges for this memory to be read-only, and any attempt by a process to change the contents of a location in this area will result in a trap (usually a segmentation violation or bus error).

The data area contains all static and global variables, initialised and uninitialised, of the program.

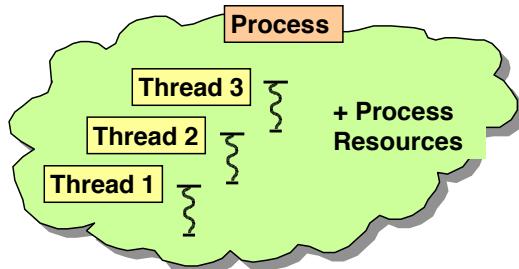
The heap area is used when more memory is allocated to the process dynamically (using malloc() and related routines).

The stack area is used to hold the stack frames of the process, which may contain return addresses from routines and automatic variables of the routines being called.

There will normally be a (very large) hole in the memory map between the heap and the stack. However, given that both areas may grow during execution, it is possible, although unlikely, that they may collide. In this case, the process will terminate with an appropriate error message.

## A Thread

- An asynchronous function call
  - Convenient method of achieving ‘concurrency’ within a process
- A thread has its own:
  - Priority
  - Register values
  - Stack
  - Local variables
  - Input queue for messages
  - Thread Local Storage



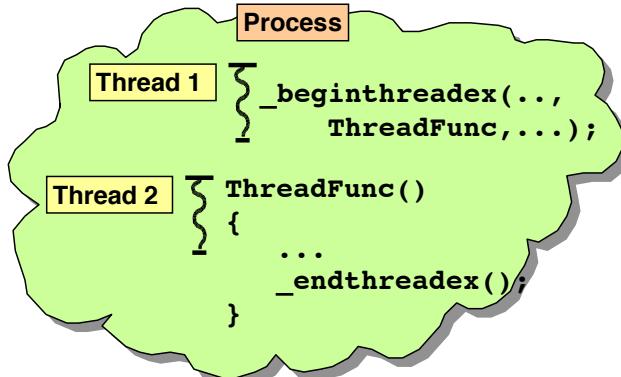
- Preemptive thread switching performed by scheduler

A thread is a dispatchable unit of execution; it is the basic entity to which the operating system allocates CPU time. Threads provide a mechanism for carrying out several programming tasks simultaneously within a process. Each thread has a priority, and switching between threads is carried out pre-emptively by the operating system scheduler. Each thread runs independently and maintains a set of data structures for saving its context while waiting to be scheduled for processing time. These structures include its own set of machine registers, its own stack and a thread environment block. A typical Win32 application will consist of many threads.

From a programming language perspective, a thread may be considered as an asynchronous function. When a normal C function is called, the flow of execution transfers from the calling environment to the function, and then returns when the function ends. But when one thread starts another, the original continues to run while the new thread executes concurrently and independently. There may thus be many copies of a function running within a process as different threads. Threads of the same process can execute any part of the program’s code, including a part being executed by another thread.

The operating system divides the available CPU time among the threads that need it. This pre-emptive multitasking means that the system allocates small slices of CPU time among competing threads. The currently executing thread is suspended when its ‘time quantum’ elapses, or if it gets pre-empted by a more important thread, allowing another thread to run. When the system switches from one thread to another it saves the context of the suspended thread and restores the saved context of the thread to be run. Although it appears that multiple threads are executing at the same time, in fact they are not, as there will only be one CPU.

## Creating Threads



- **Thread may be started:**
  - Ready to run
  - Suspended
- **Specify:**
  - Stack size
  - Arguments
  - Suspend flag
  - Thread function
- **Returns:**
  - Handle to thread
  - Thread ID

The API CreateThread() should not be used with C/C++

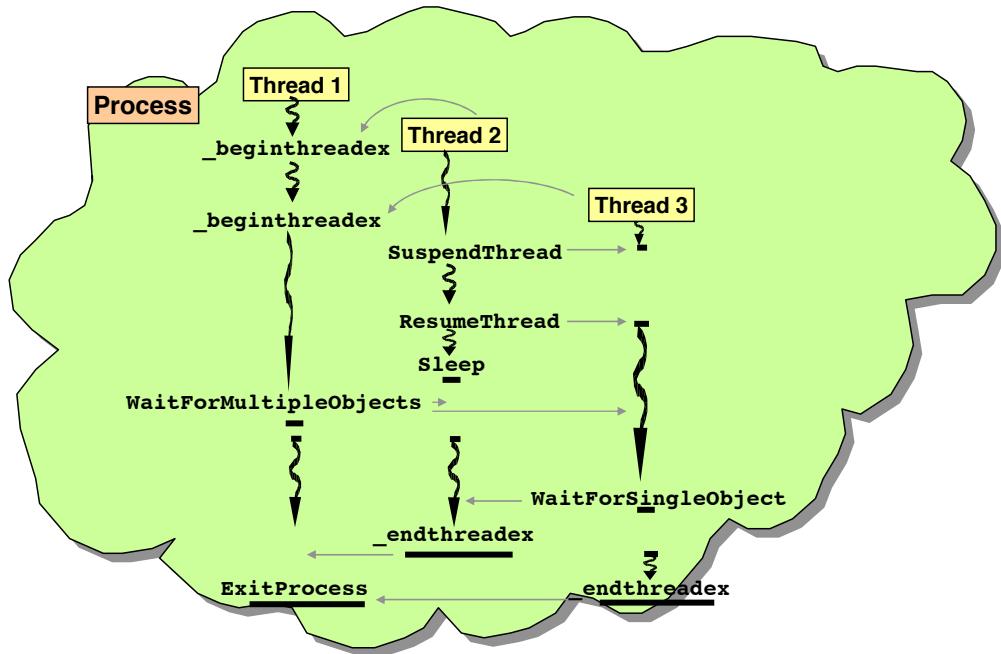
Every process has a primary thread of execution, started when the process is created. Any thread can create both processes and threads. Threads are created with less overhead than processes, although they have less protection between them. Applications are more likely to multitask by using multiple threads in the same process rather than multiple processes, unless threads with protected, private address spaces are required.

Asynchronous tasks can be carried out on behalf of a process by creating new threads of execution, using `_beginthreadex()`, specifying: (1) the size of the stack allocated from the process address space when the thread is created, and freed when the thread terminates; if 0 is specified, the default application stack size is used, i.e. that size used by the primary thread (2) the address of the thread function code to execute (3) and optionally the arguments passed to the thread function (if not specified, the thread handle cannot be inherited).

The system passes back a thread handle and ID. The size of the stack for a thread depends on whether it is a GUI or console application, and on program behaviour, e.g. re-entrancy.

No assumptions about the sequence and timing of the thread starting execution should be made. `_beginthreadex()`, returns asynchronously to inform us that a thread object has been created. This thread is waiting, along with all others in the system, to be executed by the scheduler, according to its priority; it may or may not already have been executed by the time `_beginthreadex()`, returns. This is an important issue when passing pointer arguments to a new thread; if the memory they reference exists on the stack, will arguments still be valid when the thread executes?

## Controlling Threads



322

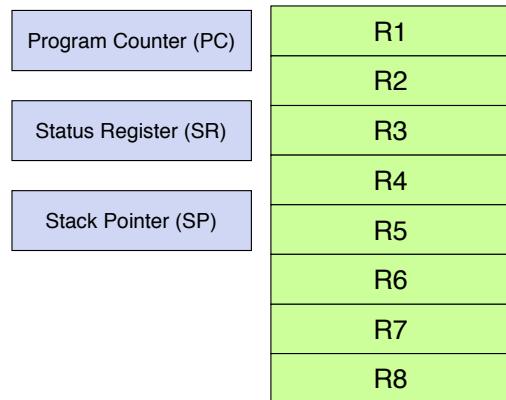
One thread can suspend or resume the execution of another by calling SuspendThread() or ResumeThread(). A suspended thread is not given CPU time by the scheduler. You can never really be sure where a suspended thread will suspend execution, so it is not a good synchronisation technique. A thread can be created suspended, to be resumed later by the creating thread, and it can use Sleep() to suspend its execution for a specified interval.

Some Win32 objects are ‘synchronisation objects’. A thread waiting on such an object is blocked if the object is not signalled. Blocked threads, like suspended threads, are not scheduled, but it is possible to specify the exact place at which the thread will wait. A thread is a synchronisation object; un signalled during its lifetime and signalled when it terminates. One thread can wait for another to terminate.

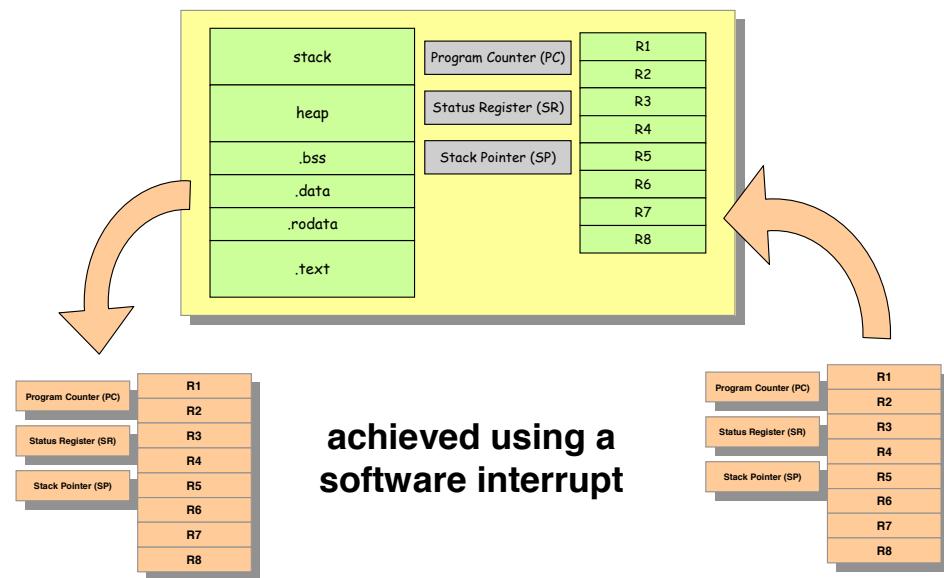
For threads of a single process that are all sharing the same resources, it is necessary to control access to these resources and to co-ordinate and synchronise thread actions. To do this, threads typically use synchronisation objects, like ‘mutexes’, ‘semaphores’ and ‘events’, to signal each other in order to synchronise their activities. These objects can be ‘waited on’, until they attain a signalled state, using APIs like WaitForSingleObject() or WaitForMultipleObjects().

## The Context Switch ...

- When a task (thread) is running it uses
  - registers
  - stack
  - private kernel context
    - pid, cpu usage, locks ...
- When kernel swaps between tasks
  - this register set needs saving and a new set installing



## ... The Context Switch



**achieved using a  
software interrupt**

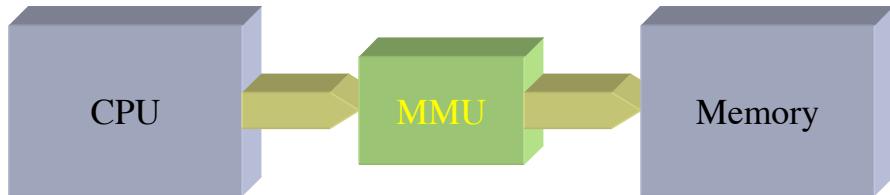


Copyright ©1994-2011 CRS Enterprises

324

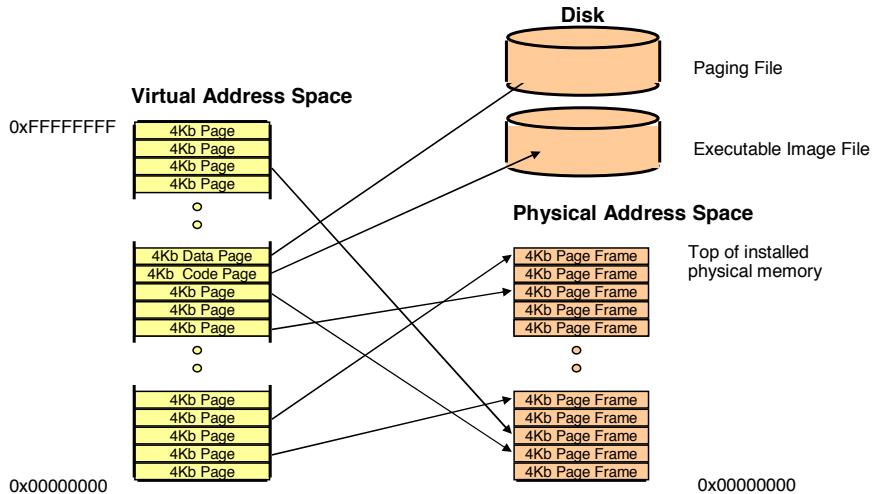
## Memory Protection

- Many larger embedded systems not make use of a
  - Memory Management Unit (MMU)



- Variants are:
  - Memory Protection Unit (e.g. ARM740)
  - Segmentation Unit (e.g. 80x86)
- Context Switch takes longer

## Paging ...



- **Valid virtual pages mapped to physical page frames**
  - Known as the process **Working set**

Because it is very inefficient to swap a byte at a time to and from disk, the virtual address space is divided up into blocks of equal sizes called pages. Similarly, physical memory is divided into blocks of the same size called page frames. The size of a page depends on the host processor. The Intel 80386, which provides comprehensive paging translation support in hardware, fixes the page size at 4 Kb. The DEC Alpha chip fixes the page size at 8 Kb, and the MIPS R4000 supports programmable page sizes between 4 Kb and 64 Kb. Programmers shouldn't therefore assume the page size is 4 Kb!

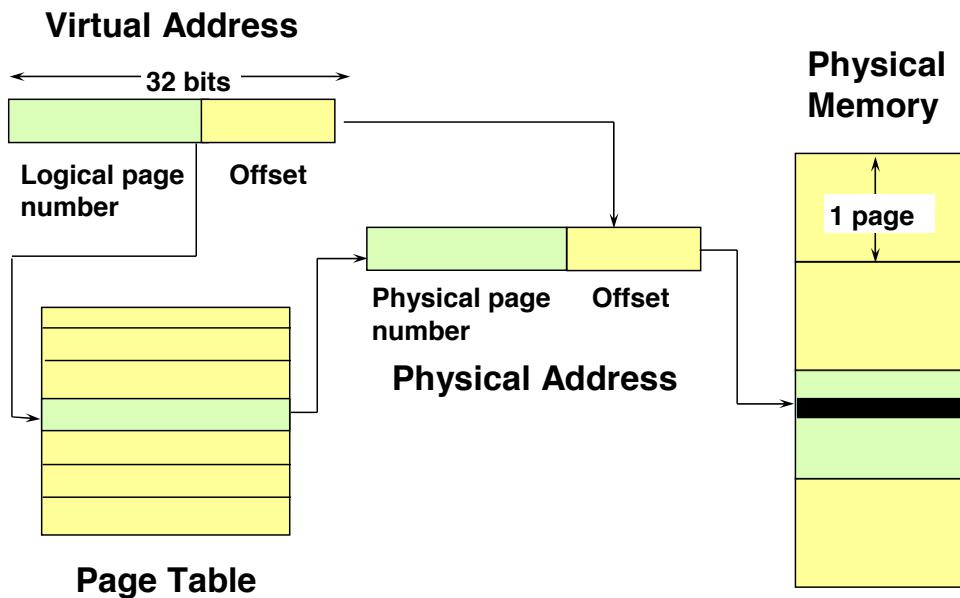
Any page in the virtual address space of a process can be in one of three states. If a page is mapped to a page frame in physical memory or to a page-sized entry in a file on disk it is said to be committed. If a page has been set aside for future use so that it cannot be re-used for any other purpose within the process, but has no associated physical page storage, it is said to be reserved. If a page is available to be reserved it is said to be free.

At any given time, a native process may only have a subset of valid pages in its virtual address space, i.e those that are mapped to page frames in physical memory. This subset is known as the working set. Pages that are temporarily stored on disk or reserved or free are known as invalid pages.

When an executing thread attempts to access an invalid page, the processor raises a particular exception called a page fault. The VMM can respond in a number of ways - see a later slide on paging policy. To respond to a page fault caused by the invalid page being on disk, the VMM will load the required page from disk into a free page frame in physical memory and re-execute the instruction which caused the fault. This activity is known as demand page loading. When the number of free page frames runs low, the VMM selects page frames to free and copies their contents to disk. This activity, known in Windows NT as paging, is transparent to the programmer.

Contd.

## Virtual and Physical Addresses



Copyright ©1994-2011 CRS Enterprises

327

Processes use virtual (or logical) addresses to access locations in their virtual memory. Basically, a virtual address can be thought of as comprising two parts: a logical page number and an offset within the page. We can do this because the logical and physical pages for a given system will be the same size and we want to translate a logical page number into the corresponding physical page number.

The number of bits used to represent the offset and page number is dependent on the implementation. For example, there must be enough bits in the "offset" portion to represent all possible locations in a page, so this will depend on the system's page size. For a system with pages of 8KB bytes, the "offset" portion of the virtual address would be 13 bits.

The logical page number is actually an index into a table known as the page table, which contains details of all the logical pages for a process. The information held in the page table includes (obviously) the address of the start of the physical page corresponding to the logical page, and also whether that page of memory can be written to (it may be part of the process text segment, in which case it is read only).

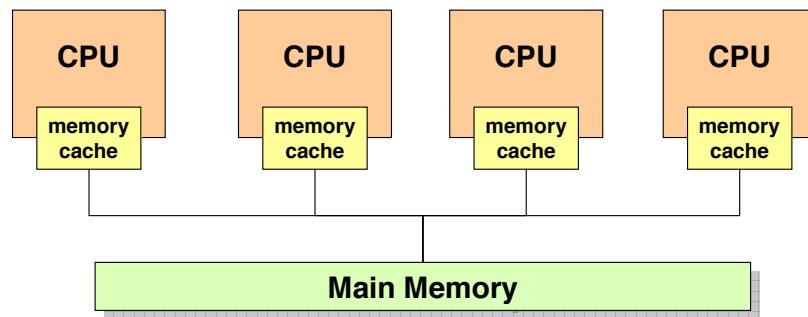
The translation from logical to physical address involves obtaining the physical page number (read from the correct page table entry) and combining it with the offset portion of the logical address to specify the location within the physical page. As part of this procedure, the access controls for the page are also checked, and if necessary a write-protect exception can be raised.

The mapping of logical to physical address is normally carried out by the memory-management-unit hardware in a system, as it clearly must be an extremely fast operation.

The page tables are part of a process's context, and the operating system arranges for the correct page tables to be loaded into the MMU whenever a process is scheduled to run.

## Multi Core Architectures

- **Shared memory architecture on multi CPU architectures**
  - each CPU has an on board cache
  - pipelining effects order of execution
- **C++ is not designed for this architecture**
  - single CPU model only
  - must use POSIX extensions
  - to be addressed in new C++ standard



Copyright ©1994-2011 CRS Enterprises

328

Modern multiprocessor systems are normally built according to an architecture similar to that shown in the slide. Multiple CPU modules share access to shared memory, using their own intelligent memory interface modules to manage this access. The memory interfaces will normally have a high speed cache to keep local copies of data and logic to perform optimisations on memory accessing patterns.

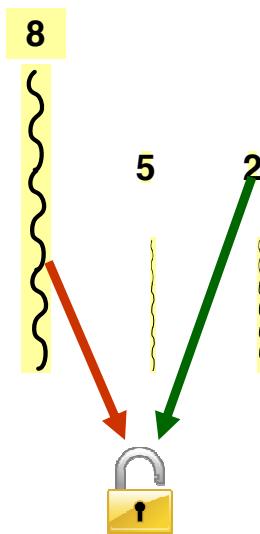
If two threads that share data are running on different CPUs, they must be sure to read data from memory rather than their own local cache to avoid using stale information. Modern memory interface units make extensive use of pipelining. With pipelining, memory accessing instructions (reads and writes) are reordered to speed up processing. This has no effect in a single threaded environment, but may cause complications with multiple CPU architectures.

It is important to realise that C++ assumes a single CPU architecture. To ensure programs function correctly in a multithreaded environment you must need to use library code that enforces additional constraints.

These issues will be addressed in the upcoming C++ standard.

## Priority Inversion

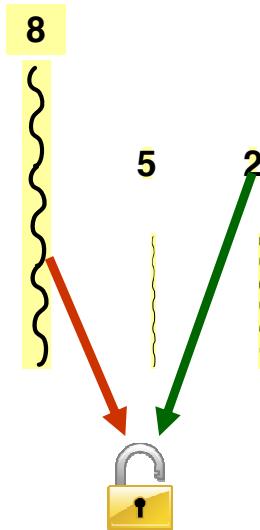
- Low priority (2) thread acquires a lock
- High priority (8) thread tries to acquire the same lock
  - blocks waiting for low priority lock to be rescheduled
- Medium priority (5) thread runs and starves out the lower priority thread
  - low priority thread never has a chance to release the lock
- High priority thread stays blocked
  - runs with the effective priority of the low priority thread



Priority inversion is where a low priority thread holds a shared resource that is required by a high priority thread. This causes the execution of the high priority thread to be blocked until the low priority thread has released the resource, effectively "inverting" the relative priorities of the two threads. If some other medium priority thread attempts to run in the interim, it will take precedence over both the low priority thread and the high priority thread.

## Solving Priority Inversion

- Java Specification has nothing to say
- Some JVMs use priority donation
  - when a high priority thread becomes blocked by a low priority thread, it donates its priority temporarily
  - the low priority thread can then run
- Java Real-Time System 2.0 (Java RTS)
  - Implementations of the RTSJ make standard Java technology more deterministic and enable it to meet rigorous timing requirements for mission-critical real-time applications.
- Consider non-blocking algorithms
  - use the new compare and swap (CAS) approach



Some JVMs use priority donation to overcome priority inversion. When a high priority thread becomes blocked by a low priority thread, it donates its priority temporarily to the low priority thread which will then be scheduled as if it were the high priority thread. Once the thread releases its lock it reverts to its original low priority.

Unfortunately, the Java Specification has nothing to say about priority inversion and hence you can't rely on all JVMs providing this facility (affects portability). This means you are tied to a particular JVM; against the spirit of "Java can run anywhere".

One solution is to use the Java Real-Time System 2.0 (Java RTS) extensions to Java. Implementations of the RTSJ make standard Java technology more deterministic and enable it to meet rigorous timing requirements for mission-critical real-time applications.

Alternatively, it may be possible to avoid priority inversion by using non-blocking algorithms (see section on the new concurrency package).

## Concurrency and Libraries

- As C is a sequential language, many C libraries are not “thread safe”, i.e. they are not re-entrant
- Three different behaviours
  - Fully re-entrant
    - e.g. `fy = cos(x);`
  - re-entrant with incorrect behaviour
    - e.g. `x = rand(); /* due to seed value */`
  - Not re-entrant and highly unsafe
    - e.g. `p = malloc(sizeof(type));`
- Thread safe libraries are slower than unsafe ones due to required locking

## Memory Mapping - Pointers



```

struct io_port
{
    unsigned char status;
    unsigned char data;
};

void main ()
{
    volatile io_port *iop = reinterpret_cast<io_port*>(0xF878);
    while (!(iop->status & 0x10))
        ; //wait for ready flag
    iop->data = 'x'; //write data
    ...
}

```

## Memory Mapping - Placement New

- **Placement new**

- note use of \*

```
struct io_port
{
    unsigned char      status;
    unsigned char      data;
    io_port() { }
};

int main()
{
    void* address = reinterpret_cast<void*>(0xF878);
    volatile io_port& myport = * new (address) io_port();
    ...
}
```

## Sequence Points

- **Sequence Points not specified by C++, but is specified by C**
  - && and ||
  - left operand of the comma operator
  - function-call operator
  - first operand of the conditional operator
  - end of a full initialization expression
  - end of an expression statement
  - conditional expression in an if, switch, while or do
  - each of the three expressions of a for statement
  - expression in a return statement
- **Compilers can reorder instructions between sequence points**
  - for optimization

## Volatile

- **Objects declared as volatile**
  - are not used in certain optimizations
  - their values can change at any time
    - the system always reads the current value of a volatile object at the point it is requested, even if a previous instruction asked for a value from the same object
    - the value of the object is written immediately on assignment

```
int volatile count;
```

- **Writes to volatile objects have Release semantics**
  - a reference to a global or static object that occurs before a write to a volatile object in the instruction sequence will occur before that volatile
- **Reads of volatile objects have Acquire semantics**
  - a reference to a global or static object that occurs after a read of volatile memory in the instruction sequence will occur after that volatile read in the compiled binary





# A3

## Optimization

- Key Components
- Scheduler
- Process map
- Paging
- Multi Core Architectures
- Memory Mapped Devices



A3

# Optimization

- **Optimization Techniques**
  - strategies
  - for embedded systems
- **Techniques from C**
  - loop unrolling
  - structure alignment
  - declaring variables
- **Techniques from C++**
  - references and pointers
  - constructors
  - exception handling



## Avoid Macros

- **Avoid #defines**
  - not part of the compiler
  - can cause unexpected bugs
- **Use const or enum**
  - C++ introduced these constructs to replace #defines
- **Don't use macros to create clever code**
  - often obfuscates code
  - usually some other mechanism better
- **Don't use macros to v=create generic code**
  - consider using templates instead

## Alignment of Structures

- **Adjust structure sizes to power of two**
  - When compiler calculates the size of an array of structures
    - shift operation for aligned structures
    - expensive multiply for non aligned structures
- **Array indexing has similar overheads**

## Place Case Labels in Narrow Range

- **If the case labels are in a narrow range**
  - compiler generates a jump table
  - rather than an if-else-if cascade
- **Jump table is much faster**
  - performance of a jump table is independent of the number of case entries in switch statement
- **Place the frequent case labels first**
  - in case jump table can't be used
  - you reduce the number of comparisons that will be performed for frequently occurring scenarios

## Minimize Local Variables

- **Minimizing the number of local variables in a function**
  - increases the chance the compiler will be able to fit them into registers
  - avoids frame pointer operations on stack
- **If the stack is not used**
  - improves performance over accessing them from memory
  - if no local variables need to be saved on the stack, the compiler will not incur the overhead of setting up and restoring the frame pointer

## Declare Variables with Innermost Scope

- **Avoid declaring all local variables in the outermost function scope**
  - forces the compiler to allocate space even if variable not used

```
int foo(char *pName)
{
    if (pName == NULL)
    {
        A a;
        ...
    }
    return ERROR;
}
```

## Reduce the Number of Parameters

- **Function calls with large number of parameters**
  - may be expensive due to large number of parameter pushes on stack on each call
- **Avoid passing complete structures as parameters**
  - use pointers and references instead

## Use References for Parameters

- **References are more efficient than copying objects**
  - you could use pointers, but syntax is less inviting
  - only need to pass the address of the object
- **References avoid overhead of CTOR and DTOR calls**
  - avoids copy CTOR
- **Pass by const reference**
  - if you don't want the function to modify objects passed
- **Don't use references for built in types**
  - not any more efficient
  - can cause problems when casts are involved
    - temporaries are created
- **Similarly consideration apply for function returns**

## Define Lightweight Constructors

- **Constructor will be invoked for every object creation**
  - many times the compiler might be creating temporary object over and above the explicit object creations in your program
  - optimizing the constructor might give you a big boost in performance
- **Arrays of objects**
  - always call the default constructor for each object in the array
- **Library code often requires a default constructor**
  - usually a good idea to provide an efficient default constructor

## Prefer Initialization over Assignment

- Using assignment can involve an unnecessary CTOR call
  - object must call CTOR before the assignment
- Try to use a CTOR to create object with sensible values

```
void foo()
{
    Complex c;
    c = (Complex)5;
}
```

```
void foo()
{
    Complex c(5);
}
```

## Use Constructor Initialization Lists

- **Always use constructor initialization lists**
  - the compiler will imply a default list if one is not given
  - can give rise to double initialization
- **Avoid trying to initialise in the body of the constructor**
  - the one exception is if object contains an array
  - it must be initialized in the body

## Consider Two-Phase Construction

- **Objects with one-phase construction are fully "built" with the constructor**
- **Objects with two-phase construction is minimally initialized in the constructor and fully "built" using a class method**
  - Frequently copied objects with expensive constructors and destructors can be serious bottlenecks and are great candidates for two-phase construction

## Be aware of Virtual Function Overhead

- **Virtual functions have overhead**
  - Every object has a V-Table pointer
    - 4 byte overhead on a 32 bit machine
  - Every class has one V-Table
    - an array of function pointers to the virtual functions
  - Each virtual function call
    - 2 extra pointer dereferences and an indirect call
- **But virtual functions are essential for polymorphism**
  - and the above overhead is often an acceptable price to pay

## Inline Small Functions

- **Use inline for small functions**
  - gives you big improvements in throughput
  - removes overhead of a function call and associated parameter passing
- **Don't inline larger functions**
  - increases memory footprint and can run out of memory
  - other function calls may end up less efficient
    - functions not on same memory page
- **Modern compilers will do a good job of inlining automatically**
  - only override after profiling provides sufficient evidence

## Static Functions are Private

- **Declaring a function as static**
  - forces an internal linkage within that file
  - improves opportunities for compiler to optimize code
  - avoids name clashes
- **Can inhibit certain optimizations**
  - such as aggressive inlining, with some C/C++ compilers
- **Consider using namespaces instead**
  - more flexible
  - C++'s replacement mechanism for static functions

## Prefer Array to Pointer Notation

- **Use array notation instead of pointer notation when working with arrays**
  - both give identical results
  - array notation often easier to understand
- **But it's hard for the compiler to optimize pointers**
  - because of possible pointers aliasing
  - array pointers are read only
  - ordinary pointers are usually declared mutable
- **Use a profiler to check which notation produces the faster code**
  - varies from processor to processor

## Unroll Loops

- **that have a small fixed loop count and a small loop body**
  - many compilers do not aggressively unroll loops
  - manually unrolling loops may be necessary
- **improves performance**
  - avoids testing the loop condition
  - loop condition overhead can be significant

## Expression Order is Significant

- When coding a complex conditional expressions
  - short circuit evaluation
  - OR terminates as soon as an operand is true
  - AND terminates as soon as an operand is false
- Complex branches
  - consisting of many Boolean expressions with logical AND and OR
  - provide great opportunities for optimization
    - if you know which AND and OR conditions are more likely to occur
- Use to avoid null pointers

```
// if ptr == 0, *ptr is never evaluated
if(ptr != 0 && *ptr > 100)
{
    // ...
}
```

## Limit Exception Handling

- **Exceptions are a great way to deal with unexpected errors**
  - but they're expensive
  - performance is degraded
    - can be very slow with some compilers
    - modern compilers are much better
  - memory footprint is increased
    - up to 10% extra code generated
- **Embedded systems usually avoid exception handling**
  - but more and more core libraries make use of exceptions
  - must stop new and delete throwing exceptions
    - use operator new with the no\_throw specification
- **If you use exceptions**
  - use throw lists to define what may be thrown

## Avoid Runtime Type Identification

- **Runtime type identification (RTTI)**
  - doesn't add much to C++
  - not like reflection in Java
  - class name appended to V-Table
    - in a `type_info` object
    - class must define at least one virtual function
    - `dynamic_cast` operator relies on RTTI
- **Very few programs need to use RTTI**
  - don't use it unless you need it
  - overhead is not that significant

## Prefer stdio to iostream

- **iostream is the "Object Oriented" solution**
  - but it's awkward to use
  - difficult to handle input errors
  - formatting output is non trivial
- **stdio is not type safe**
  - but it's simple to use
  - good at handling invalid input
  - good at formatting output
  - more efficient than iostream
- **printf is ever popular**
  - it's fast
  - it's easy to use
  - easier to read than long lines of << operators

## Optimization Strategies that Bomb

- **Don't assume some operations are faster than others**
  - always benchmark everything
  - use a profiler
- **Don't optimize too early**
  - "premature optimization is the root of all evil"
- **Worrying about performance before concentrating on code correctness?**
  - write the code without optimizations first
  - let performance issues guide your design, data structures, and algorithms
  - don't let performance affect every aspect of your code
  - only a small percentage of the code requires optimization

## Use Prefix Operators

- Overload **++** and **- -** prefix operators are more efficient
  - can return by reference
  - don't need to create a snapshot

prefix

```
Time& Time::operator++()
{
    // inc *this
    return *this;
}
```

postfix

```
Time Time::operator++(int)
{
    Time snapshot = *this;
    // inc *this
    return snapshot;
}
```

## Return Value Optimization

- Returning a temporary can be more efficient than returning a named object
  - compilers can avoid "invisible" creation and destruction of temporaries when function returns an object by value
  - known as "return value optimization"

```
Time doit( )
{
    // ...
    return Time(5, 30);
}
```



```
Time doit( )
{
    Time t(5, 30);
    // ...
    return t;
}
```



# A4

## Real Time Design Patterns

- Thread Affinity
- Thread Pools
- Futures, Exchangers, RTTI
- Synchronization Pattern
- Barrier, Latch
- Monitor Object
- Active Object
- Half Sync/Half Async
- Leader/Follower



**A4**

In this chapter we explore some of more important Real Time design patterns.

## Thread Affinity

- **Create a thread for each client request**
  - simple to program, but
  - classified as an anti-pattern
- **What is wrong with thread affinity?**
  - threads consume valuable O/S resources
  - 1 thread per client doesn't scale well
    - thread exhaustion
    - too much context switching
  - threads not all working at same rate
    - some threads can become inactive for long periods
- **Consider Thread Pools**

Thread affinity is where we create a thread for each client request. This pattern is simple to program, but is classified as an anti-pattern for several reasons. So what is wrong with thread affinity?

Threads consume valuable O/S resources and therefore 1 thread per client doesn't scale well: it is possible to run out of threads if there are too many clients. Furthermore, the more threads we have, the more context switching will occur. Not all clients will make requests at the same rate and hence not all threads will be working at same rate and indeed some threads may be idle for long periods.

Instead, consider Thread Pools.

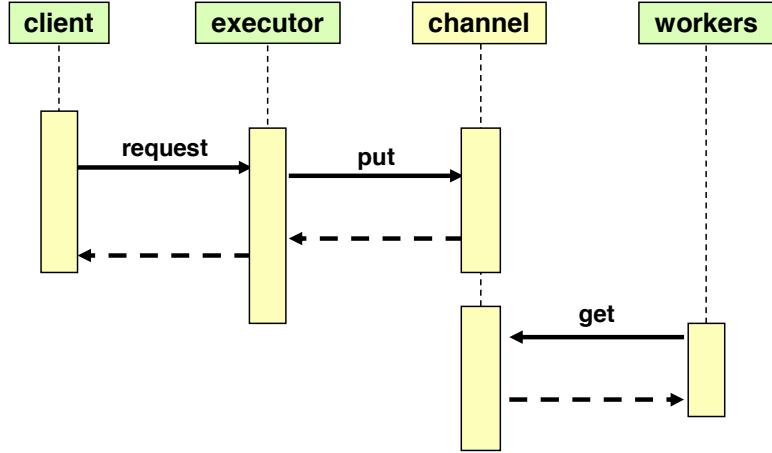
## Thread Pools ...

- **What is a Thread Pool?**
  - a collection of co-operating threads, where each thread is decoupled from client requests
- **The Executor**
  - one thread (executor) is responsible for receiving requests from clients and queuing the requests on some kind of channel
- **Worker threads**
  - worker threads process execute many unrelated tasks (avoiding client affinity) by extracting tasks taken from the channel
- **Channel**
  - the channel is the communication medium between the executor and the worker threads
  - the channel must provide synchronization between cooperating threads
  - must be an O/S primitive

A Thread Pool is a collection of co-operating threads, where each thread is decoupled from client requests. This allows threads to service from any client and spread the workload. The thread pool usually employs an executor, where one thread is responsible for receiving requests from clients and queuing the requests on some kind of channel. Worker threads then process these requests asynchronously.

The channel is the communication medium between the executor and the worker threads and breaks thread affinity. However the channel must provide synchronization between cooperating threads.

## ... Sequence Diagram



Copyright ©1994-2011 CRS Enterprises

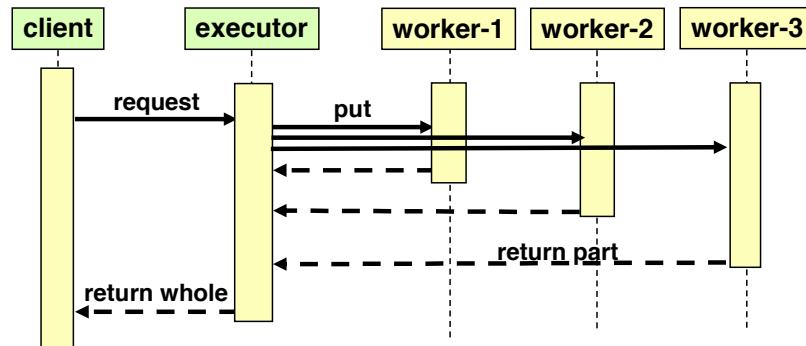
368

This sequence diagram shows the executor putting client requests on the channel and returning control to the client immediately. The client is then decoupled from the request which is processed later by a worker thread.

Note that the decoupling of the client and worker threads in this pattern works best if the worker does not need to return a result to the client (fire and forget). If a response is required, an additional pattern will need to be employed.

## Futures ...

- Variation on executor and worker thread pattern
- Caller asks executor to perform a computational task
  - Executor divides up the task and hands off to a set of Worker threads
  - Each worker calculates part of the final total and returns its result to the executor
  - Executor gathers together results and calculates the final total
  - Executor returns result to caller



Copyright ©1994-2011 CRS Enterprises

369

The Futures pattern is a variation on executor and worker thread pattern. The caller asks the executor to perform a computational task and the executor divides up the task and hands off to a set of worker threads. Each worker calculates part of the final total and returns its result to the executor. The executor gathers together results and calculates the final total and returns the result to caller. The key part of this pattern is that the executor handles all the locks relieving the client code from worrying about synchronization.

## ... Futures

```

int main()
{
    MyThread t1(squares, 1, 2);
    MyThread t2(squares, 3, 4);
    MyThread t3(squares, 5, 6);

    Futures futures;
    futures.request(&t1);
    futures.request(&t2);
    futures.request(&t3);
    futures.start();

    stringstream result;
    result << futures.getResult();

    MessageBoxA(0, result.str().c_str(), "Main", MB_OK);
}

```

The diagram illustrates the execution flow of the provided C++ code. It is divided into three vertical sections by horizontal lines. Red arrows point from the right side of each section to specific lines of code in the main function:

- The top section, labeled "queue up requests", contains the first three `futures.request(&t1);`, `futures.request(&t2);`, and `futures.request(&t3);` calls.
- The middle section, labeled "start calculation", contains the `futures.start();` call.
- The bottom section, labeled "get result", contains the `result << futures.getResult();` call.

In the test program above the `MyThread` class constructor takes 3 parameters. The first parameter is a function pointer to a task that the thread will perform. In this case the thread will calculate the squares of numbers between the limits specified by parameters 2 and 3.

The threads do not calculate these values immediately. Instead, each thread object is queued by the `request` method of the `Futures` object. When all the tasks have been queued, the `Futures` object is told to start the calculations. The `Futures` object starts each of the threads and each of the threads perform their part of the squares calculation.

Finally, the `Futures` object gathers the results from each of the threads and combines them into a single result. As we will see on the next slide, the `Futures` object waits for each of the threads to complete their calculations before aggregating the result. However the complexities of the synchronization is hidden inside the `Futures` class.

## ... Futures

```

void Futures::request(Thread* t) {
    pool.push_back(t);
}

void Futures::start() {
    for(unsigned i = 0; i < pool.size(); i++) {
        pool[i]->start();
    }
}

int Futures::getResult() {
    HANDLE* handles = new HANDLE[pool.size()];
    for(unsigned i = 0; i < pool.size(); i++) {
        handles[i] = pool[i]->getHandle();
    }
    WaitForMultipleObjects(3, handles, true, INFINITE);
    int result = 0;
    for(unsigned i = 0; i < pool.size(); i++) {
        result += pool[i]->result();
    }
    return result;
}

```

Copyright ©1994-2011 CRS Enterprises

371

Inside the Futures class, the MyThread objects are pushed onto the thread pool. The start method initiates each of the threads and they enter their run methods (see next slide). The threads each perform part of a calculation.

In the getResult method, the Futures object waits for each of the threads to complete their tasks (using WaitForMultipleObjects()). When all the threads have finished, each result is collected and aggregated.

## ... Futures

```

typedef int (*FP)(int low, int high);
class MyThread : public Thread
{
public:
    MyThread(FP fp, int low, int high)
        : low(low), high(high), fp(fp) {}
protected:
    int run();
private:
    int low;
    int high;
    FP fp;
};

int MyThread::run()
{
    int result = fp(low, high);
    stringstream s;
    s << result;
    MessageBoxA(0, s.str().c_str(), "Thread", MB_OK);
    return result;
}

```

*calculation to be performed*

```

MyThread t1(squares, 1, 2);
MyThread t2(squares, 3, 4);
MyThread t3(squares, 5, 6);

```

```

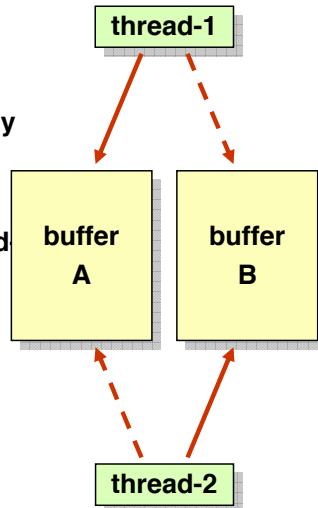
int squares(int low, int high) {
    int result = 0;
    for(int i = low; i <= high; i++) {
        result += i * i;
    }
    return result;
}

```

The MyThread class represents the unit of work to be performed by the thread. In its constructor the first parameter is a function pointer that represents the calculation to be performed, in this case to calculate squares between low and high. The actual calculation is performed in the thread's run method.

## Exchangers

- **Threads share swing buffers**
  - buffers not swapped until both threads are ready
  - each thread signals O/S when it is ready to swing
- **Swing buffering technique**
  - streaming real time data through thread 1
  - process data with thread-2
  - in streaming situations, thread-1 must swing immediately on filling the buffer
    - thread-2 must complete before thread-1



The Exchanger pattern is where two threads share swing buffers. A thread fills one buffer while the other thread analyses the data in the other buffer. Usually the thread performing the analysis can only work with a complete buffer - hence the requirement of two buffers. Once both threads have completed their work the buffers are swapped and the process repeats.

The swing buffering technique is often used when processing streaming real time data through with one thread and processing the data with the other thread. In streaming situations, the first thread must swing immediately on filling the buffer and hence the second thread must complete before the first thread.

## Resource Acquisition is Initialization ...

- How can we ensure we release resources correctly
  - patterns to control resource acquire and release as pairs
  - very similar to initialization of objects

```
class Locker
{
public:
    Locker(HANDLE h) : hMutex(h) {
        WaitForSingleObject(hMutex, INFINITE);
    }
    ~Locker() {
        ReleaseMutex(hMutex);
    }
private:
    HANDLE hMutex;
};
```

- Use a class that
  - acquires in its constructor
  - releases in its destructor
  - when the object goes out of scope, resource is automatically released

A common theme in design is how can we ensure we release resources correctly. The classic memory leak problem has plagued designers and developers alike. However, patterns to control resource acquisition and release are quite simple to formulate and are usually based on the initialization of objects and cleanup of objects via constructors and destructors.

The pattern uses a class that acquires a resource in its constructor and releases the resource in its destructor. This means that when a stack based object goes out of scope, the resource is automatically released with no possibility of error.

This pattern is known formally as "Resource Acquisition is Initialization".

## ... Resource Acquisition is Initialization

- Use stack frame to control locking

```

HANDLE hMutex;

int main( )
{
    CreateMutex();
    {
        Locker locker(hMutex);
        _beginthread(ThreadFunc, 8192, "Thread 1");
        _beginthread(ThreadFunc, 8192, "Thread 2");
        _beginthread(ThreadFunc, 8192, "Thread 3");
    }
}

```

```

void CreateMutex( ) {
    hMutex = CreateMutex(
        0,           // security
        false,        // initial owner
        "MyMutex"   // name
    );
    if (hMutex == 0) throw "...";
}

```

Here we use a stack based Locker object to acquire a Mutex lock by virtue of its constructor being called. The Mutex is automatically released as the Locker object goes out of scope.

One minor drawback of this design is that you hold a lock until the end of the method call even though you might prefer to release it earlier. This deficiency is easily overcome by introducing an inner code block { .. } thereby reducing the scope and lifetime of Locker object.

## Synchronization Pattern ...

- **Java style locking**
  - use the synchronize pattern
  - C++ has no equivalent construct
- **Use if statement to**
  - implicitly declare a lock
  - provide a cast to convert to bool
  - that always is true
  - automatically removed at end of if
- **Now wrap up in a macro**
  - to make it look like Java

```
Lock::operator bool( ) {
    return true;
}

#define synchronized(cs) if(Lock lock(cs))
```

```
void Resource::Update( ) {
    synchronized(cs) {
        x++;
        y++;
    }
}
```

```
void Resource::Update( ) {
    if(Lock lock = cs) {
        x++;
        y++;
    }
}
```

The Synchronize pattern provides synchronization for a block of code. This pattern is used extensively in Java, but C++ has no equivalent construct.

To implement this pattern in C++ we first need a Lock class that utilizes RAII. The idea is to create a stack based Lock object that will be automatically cleaned up at the end of the code block. We achieve this using a little trick, as follows.

We surround the code block to be synchronized by a dummy if statement. The reason we use an if statement is purely to create the stack based Lock object; the if statement must always evaluate to true to make sure the block executes. At the start of the if statement, the constructor of the Lock object acquires the lock and the destructor is called at the end of if statement and releases the lock.

Note the overloaded "operator bool" method that always returns true - this makes sure the if statement executes.

The #define allows us to make our C++ code look like Java.

## Deadlock ...

- **getX() and setX() are protected with a lock**
  - data is protected
  - but threads can deadlock in Swap!

```
Cell::Cell(int x) : x(x)
{
    hMutex = CreateMutex(0, false, 0);
}

void Cell::Swap(Cell other)
{
    int lhs = getX();
    int rhs = other.getX();
    setX(rhs);
    other.setX(lhs);
}
```

```
class Cell
{
public:
    void Swap(Cell);
    int getX() {
        int rv;
        Locker locker(hMutex);
        rv = x;
        return rv;
    }
    void setX(int newX) {
        Locker locker(hMutex);
        x = newX;
    }
private:
    int x;
    HANDLE hMutex;
};
```

Copyright ©1994-2011 CRS Enterprises

377

When working with threading, care must be taken to avoid deadlock. Sometimes code that can cause deadlock looks innocuous, but has a hidden flaw.

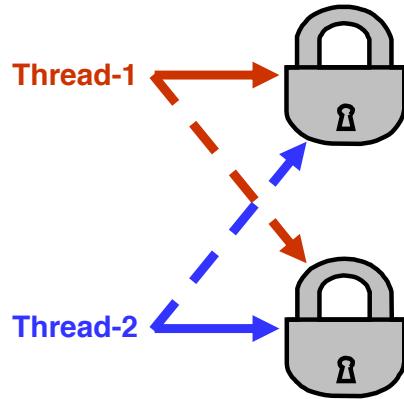
It is well known that to avoid deadlock, threads should always acquire locks in the same order. So consider the example above where the methods `getX()` and `setX()` are protected with a lock object, based on a common mutex. Although it looks as though there is only one mutex involved in this code, on careful examination we realize that each `Cell` object has its own mutex. With two different mutexes we know that deadlock is possible if the locks are not always acquired in the same order.

## ... Deadlock

- **Each Cell object maintains its own Mutex**
  - and therefore two mutexes are used
- **Threads acquire locks in reverse order**
  - deadlock possible!

```
void ThreadFunction1(void*) {
    ....
    c1.Swap(c2);
}

void ThreadFunction2(void*) {
    ....
    c2.Swap(c1);
}
```



Now consider the Swap method. This method tries to swap two Cell objects (\*this and other). What will happen if one thread executes the code

**c1.Swap(c2)**

and another thread executes

**c2.Swap(c1)**

at the same time. The first thread will acquire the lock for c1 before that for c2. Clearly the other thread acquires the locks in the opposite order. Eventually this is bound to lead to deadlock!

## ... Deadlock

- **Use resource ordering to fix problem**

- locks associated with memory address
- and can now be applied in order

```
void Swap(Cell other)
{
    if(this == &other) return;

    if(this > &other)      ← alias check
        return other.Swap(*this); ← associate lock with memory address
                                always acquire in memory order

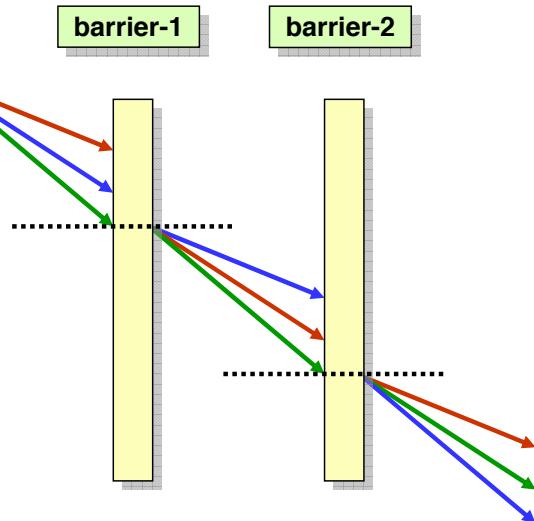
    int lhs = getX();
    int rhs = other.getX();
    setX(rhs);
    other.setX(lhs);
}
```

One way of resolving the problem is to use resource ordering. Since each Cell object has its own lock, we can use the memory address of the object to determine the order in which we apply the locks. This way we acquire a lock of the object in lower memory first.

In the above code, check if the address of \*this is greater than the address of other. If it is we are in danger of acquiring the locks in the wrong order, so we call the same method with the arguments reversed. If the check fails it is safe to proceed.

## Barrier Pattern

- **Barriers temporarily halt the execution of a thread**
  - until the other threads have caught up
- **Useful for iterative tasks**
  - threads work through tasks in stages, synchronising at the start of each stage
  - use a cyclic barrier if stages go on for ever

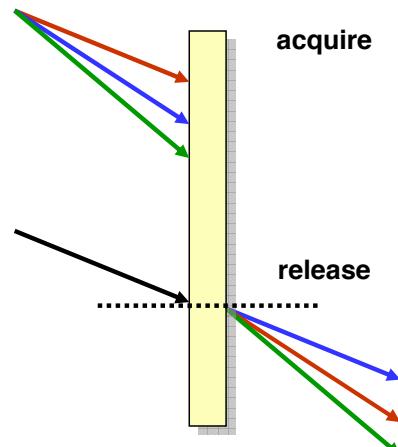


A Barrier lock temporarily halts the execution of a set of threads until all other threads have caught up. The barrier lock is akin to a counting semaphore - the barrier is prepared with an initial count and the count is decremented each time a new thread reaches the barrier (it gets blocked). When the last thread arrives at the barrier, the count falls to zero and all waiting threads are released.

The pattern is useful for iterative tasks where threads work through tasks in stages, synchronising at the start of each stage. Use a cyclic barrier if stages repeat for ever.

## Latch Pattern

- Latching variables have two states
  - acquired and released
- Latches are created in the acquired state and once released can never be reset
  - used for initialization
  - one thread will release a latch when all initialization has been completed
  - worker threads can wait on the latch to ensure that don't start too soon
- Countdown latches
  - state doesn't change until a fixed number of release calls have been made



The latch pattern is used to detect completion of initialization.

A latching variable can only have two states: acquired and released. Latches are created in the acquired state and once released can never be reset. Worker threads test the latch and are blocked if the latch is already acquired (corresponding to the state - initialization still in progress). The thread performing the initialization releases the latch when all initialization has been completed. The worker threads wait until the latch is released and then can proceed knowing initialization is complete.

Countdown latches are an extension of this concept; several initialization threads must decrement the countdown latch before it is safe for worker threads to proceed.

## More Real Time Patterns

- **Monitor Object**
  - relieves client threads of concerns about synchronization
- **Active Object**
  - decouples client requests from their activation
- **Half Sync/Half Async**
  - decouple high level synchronous services from low level asynchronous I/O
- **Leader/Follower**
  - pick a leader thread from a thread pool to listen for events

### Monitor Object

**Internal synchronizes of client threads that access a monitor object, relieving client threads of concerns about synchronization**

### Active Object

**Decouples client requests from their activation by executing an operation in a separate thread from the calling thread.**

### Half Sync/Half Async

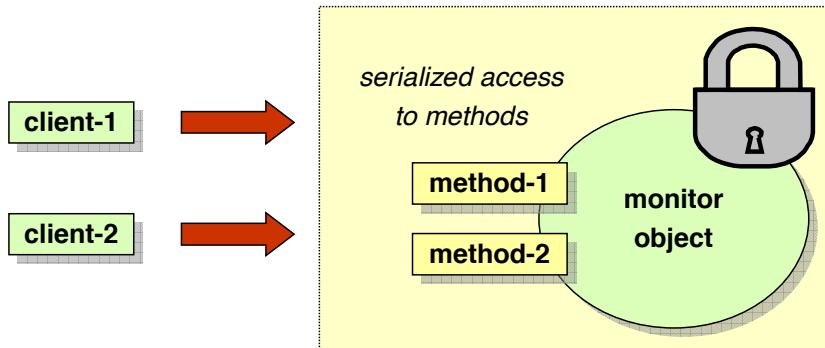
**Decouple high level synchronous services from low level asynchronous I/O using a queuing layer**

### Leader/Follower

**Using a thread pool to process events, pick a leader thread to listen for events and use the remaining threads to process events**

## Monitor Object

- A monitor object synchronizes two or more threads that use it as shared resource
  - using an internal Mutex
  - ensures serial access to its methods
  - monitor handles blocking calls transparently.



Copyright ©1994-2011 CRS Enterprises

383

A monitor object synchronizes two or more threads that use it as shared resource using an internal Mutex. This ensures serial access to its methods and relieves the client threads of concerns about synchronization. The monitor handles blocking calls transparently.

# Monitor Object Pattern

- **Motivation**

- access to shared objects requires synchronization
- get the object itself to provide the necessary synchronization internally
  - client threads use the object without worrying about synchronization

- **Consequences**

- thread access to the monitor object's state is serialized
- calling thread acquires the monitor's lock blocking other threads
- if a thread has to block whilst accessing the monitor's lock
  - the monitor is responsible for suspending that thread
  - temporarily releasing the lock
  - when the thread unblocks the thread will reacquire the lock
- if several monitor objects interact using call-backs, it is possible to introduce deadlocks.

When using a shared object (resource) it is important to synchronize access to the objects state. If the object itself provides the necessary synchronization internally, client threads will be able to use the object without having to worry about synchronization themselves. Such an object is called a monitor object.

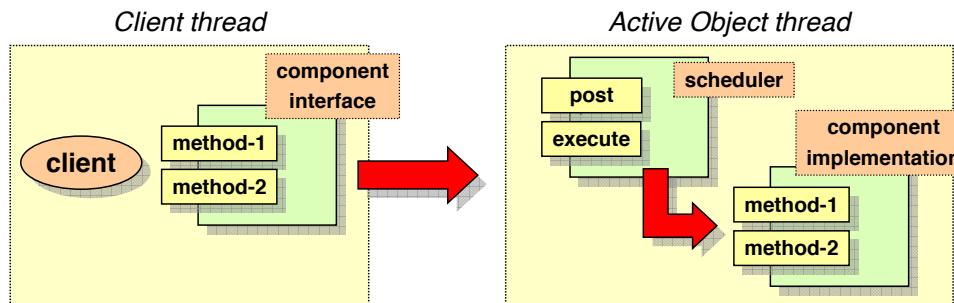
Only one thread will be able to access the monitor object's state at any given moment. The calling thread will acquire the monitor's lock and all other threads will be temporarily blocked if they attempt to access the object.

If a thread has to block whilst accessing the object, the monitor is responsible for suspending that thread and temporarily releasing the lock (first making sure its state is consistent). When the thread unblocks the thread will reacquire the lock.

On the downside, access is serialized, even if threads are not modifying the state. Another potential problem is that if several monitor objects interact using call-backs, it is possible that the system will deadlock.

## Active Object

- **Active Object decouples client requests from their activation by executing an operation in a separate thread from the calling thread.**
- **As an enhancement, a scheduler can be introduced on the active thread to prioritize requests according to requirements**



Copyright ©1994-2011 CRS Enterprises

385

The Active Object decouples client requests from their activation by executing an operation in a separate thread from the calling thread. The idea is to decouple the call from its execution.

As an enhancement, a scheduler can be introduced on the active thread to prioritize requests according to requirements.

## Active Object Pattern

- **Motivation**

- decouple the calling thread from the execution thread
- improves responsiveness
- client doesn't need to wait for the call to be completed
- Fire and Forget

- **Consequences**

- client code and service requests run concurrently
- clients are not blocked
- scheduler allows prioritization of requests
  - request can be actioned in a different order from which they were received
- Active Object is a heavyweight request handling infrastructure
  - may introduce unacceptable performance penalties

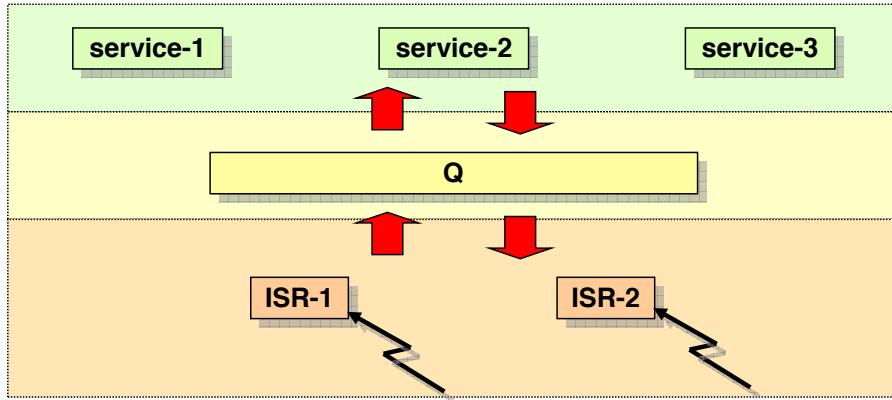
In designing concurrent systems responsiveness is a key consideration. If a high priority client invokes a method call and is blocked by the call, responsiveness suffers. Sometimes the client is not interested in waiting for the call to be completed (Fire and Forget) and merely wants to post the request for execution. Even if the client is interested in a result, it may be better to return later to pick up the result.

Client code and service requests run concurrently; clients are not blocked. Using a scheduler with the active object means that requests can be prioritized and actioned in a different order from which they are received. The scheduler can incorporate synchronization constraints, thereby alleviating the client of this responsibility.

It should be noted that Active Object is a heavyweight request handling infrastructure which may introduce unacceptable performance penalties for systems requiring rapid responses.

## Half Sync/Half Async

- **Decouples high level synchronous services from low level asynchronous I/O**
  - simplify programming effort without degrading execution efficiency
  - introduce a decoupling queuing layer



Copyright ©1994-2011 CRS Enterprises

387

This pattern decouples high level synchronous services from low level asynchronous I/O to simplify programming effort without degrading execution efficiency. A decoupling queuing layer is introduced to mediate between the synchronous and asynchronous parts of the system.

## Half Sync/Half Async Pattern

- **Motivation**

- concurrent systems perform synchronous and asynchronous operations
  - decouple using a queuing layer

- **Consequences**

- separating synchronous and asynchronous takes
    - makes system easier to understand
    - asynchronous responsiveness is not degraded by synchronous services
  - queuing system makes it easier to prioritize messages
  - but, all data must be copied to pass through the queuing system

Concurrent systems often perform both synchronous and asynchronous operations. Low level interrupt driven devices use asynchronous operations for efficiency and these need to be integrated with higher level synchronous services.

This integration is best achieved by decomposing the system into two separate layers and providing a queuing layer for communication between the layers.

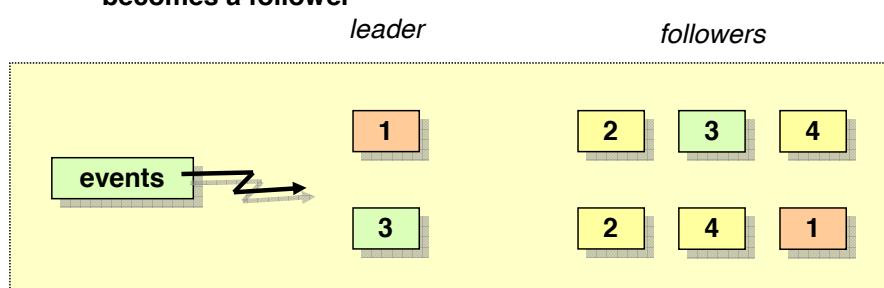
Separating layers for synchronous and asynchronous operations using a queuing layer makes comprehension of the system much easier. Synchronous services are decoupled from asynchronous operations and hence asynchronous responsiveness is not degraded by synchronous services.

A queuing system makes it easier to prioritize messages according to type.

On the downside, all data must be copied to pass through the queuing system.

## Leader/Follower

- **Use thread pool to process events**
  - one thread (**leader**) listens for events
  - the remaining threads (**followers**) process events
- **When the leader detects an event**
  - promotes a waiting thread to become the new leader
  - processing the request itself
  - **becomes a follower**



Use a thread pool to process events. Use only one thread at a time, the leader, to listen for events and use the remaining threads, the followers to process events. When the leader detects an event, it promotes a waiting thread to become the new leader before processing the request itself

## Leader/Follower Pattern

- **Motivation**

- thread pools are necessary to distribute work uniformly amongst worker threads
  - threads need to be able to listen for events
  - and process events

- **Consequences**

- no manager thread is required to coordinate the thread pool
  - threads run independently in the pool
  - do not exchange data or control information
- design works well in systems where the unit of work is short

Thread pools are necessary to distribute work uniformly amongst worker threads and avoid problems like client affinity. However, allocating work to individual threads in the pool is a difficult task. Threads need to be able to listen for events and subsequently process them. This pattern makes the design decision that only one thread will listen; in order to reduce complexity of the thread pool software.

No manager thread is required to coordinate the thread pool. Threads run independently in the pool and do not exchange data or control information.

The design only works well in systems where the unit of work is of short duration.







