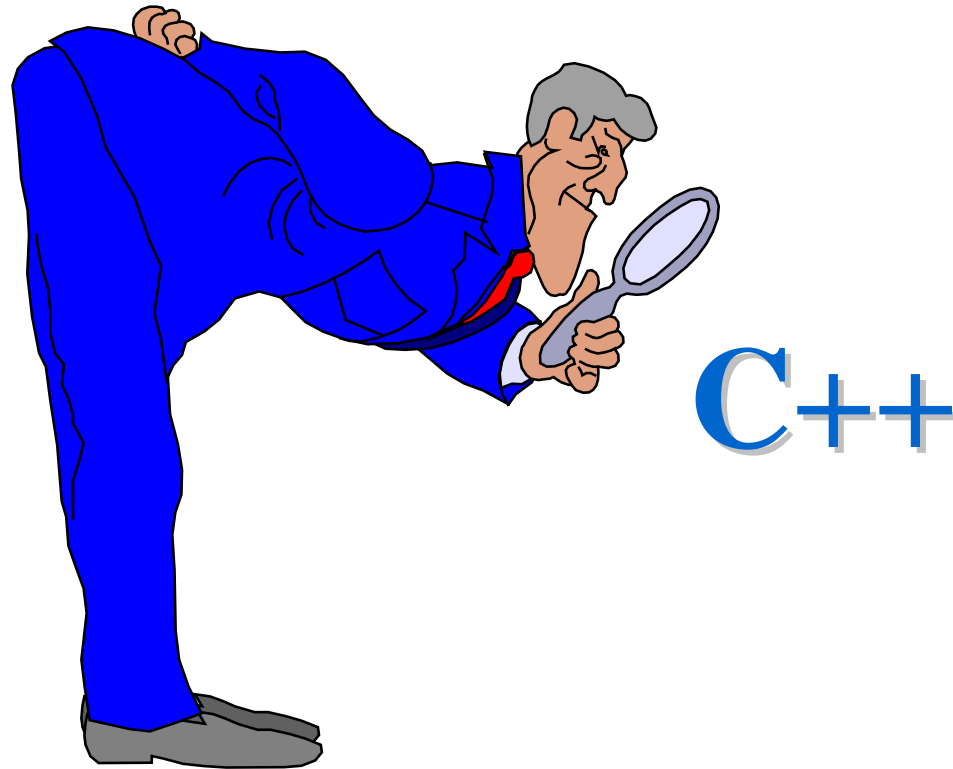


The C++ Language



C++ - Introduction

- ◆ In this section the C++ language will be presented.
- ◆ Not all the syntactical details of the language will be described, once again the focus will be on semantics.
- ◆ A step-by-step approach, from simple topics to more complex ones, will be used.
- ◆ The previous sections on Principles and C are a prerequisite for understanding this section on C++.

C++ - “A Better C”

- ◆ ANSI C and the C subset of C++ are not the same.
- ◆ Hereafter some of the most relevant differences are listed.

C++ - “A Better C”

(cont)

ANSI C

- ◆ void foo(void);
- ◆ a void * pointer can be assigned to a non void * pointer without a cast
- ◆ malloc, free (see pitagora.c)
- ◆ no alternative names for objects (I.e. references)
- ◆ type checking is better than in non ANSI C
- ◆ functions are identified via their names

C++

- ◆ void foo();
- ◆ a void * pointer cannot be assigned to a non void * pointer without a cast
- ◆ new, delete (see pitacc.cxx)
- ◆ reference, allowing calls-by-reference (see swapcc.cxx)
- ◆ type checking is better than in ANSI C
- ◆ functions are identified via their signature, operation polymorphism (see signatu.cxx)

C++ - The Concept of Class

◆ Syntax

```
<classkey> <classname> [<:baselist>] { <member list> }
```

- <classkey> is either a **class**, struct, or union.
- <classname> can be any name unique within its scope.
- <baselist> lists the base class(es) that this class derives from.
<baselist> is optional
- <member list> declares the class's data members and member functions.

◆ Description

- Use the class keyword to define a C++ class.
- Within a class the data are called **data members** the functions are called **member functions**.

◆ Example

```
class Line {  
    int x1,y1;  
    int x2, y2;  
    void display();  
};
```

C++ - The Concept of Class - Before C++

- ◆ The need for classes, I.e. object oriented mechanisms, derived from some previous experience in developing and maintaining very complex systems (e.g. emacs, X-11, etc.), some of these needs were:
 - being able to hide/encapsulate the details of some portions of the code to the others;
 - being able to develop generic pieces of code, capable of handling different objects, types without having to be modified;
 - being able to modify at run time the semantics associated to a given syntactic construct (dynamic binding).
- (See c2cpp0.c and c2cpp1.c)

C++ - Access Control

- ◆ Each declaration of a member can be pre-fixed with the following clauses:
 - **public** - the member(s) can be accessed by any function.
 - **protected** - the member(s) can be accessed by member functions and friends of the class in which it was declared, and by classes derived from the declared class.
 - **private** - member(s) can be accessed only by member functions and friends of the class in which it is declared.
- ◆ Members of **struct and unions** are **public** by default.
- ◆ Members of **classes** are **private** by default.
(See example `ac_cnt.cxx`)

C++ - Access Control

(cont)

- ◆ The keyword `const` can be used to mark data members as read-only.
- ◆ C++ extends `const` to include classes and member functions. In a C++ class definition, use the `const` modifier following a member function declaration. The member function is prevented from modifying any data in the class.
- ◆ A class object defined with the `const` keyword attempts to use only member functions that are also defined with `const`. If you call a member function that is not defined as `const`, the compiler issues a warning that the a non-`const` function is being called for a `const` object.

C++ - Static Members

- ◆ In a class, data and member functions can be declared **static**. Only one copy of the static data exists for all objects of the class.
- ◆ A static member function of a global class has external linkage. A member of a local class has no linkage. A static member function is associated only with the class in which it is declared. Therefore, such member functions cannot be virtual.
- ◆ Static member functions can only call other static member functions and only have access to static data. Such member functions do not have a `this` pointer (see example `static.cxx`)

C++ - this

- ◆ Non static member functions operate on the class type object they are called with. For example, if `x` is an object of class `X` and `f()` is a member function of `X`, the function call `x.f()` operates on `x`. Similarly, if `xptr` is a pointer to an `X` object, the function call `xptr->f()` operates on `*xptr`. But how does `f` know which instance of `X` it is operating on? C++ provides `f` with a pointer to `x` called **this**. **this** is passed as a hidden argument in all calls to non static member functions.
- ◆ **this** is a local variable available in the body of any non static member function. **this** does not need to be declared and is rarely referred to explicitly in a function definition. However, it is used implicitly within the function for member references. If `x.f(y)` is called, for example, where `y` is a member of `X`, `this` is set to `&x` and `y` is set to `this->y`, which is equivalent to `x.y`.

C++ - Inline

◆ Syntax

```
inline <datatype> <class>_<function> (<parameters>)  
    { <statements>; }
```

◆ Description

- Use the inline keyword to declare or define C++ inline functions, which are “macro-expanded” in the code.
- Inline functions are best reserved for small, frequently used functions.

See example `inline.cxx` (and `inline.s`).

C++ - Constructors

- ◆ Constructors are distinguished from all other member functions by having the same name as the class they belong to. When an object of that class is created or is being copied, the appropriate constructor is called implicitly.
- ◆ Constructors for global variables are called before the main function is called. When the `#pragma startup` directive is used to install a function prior to the main function, global variable constructors are called prior to the startup functions.
- ◆ Local objects are created as the scope of the variable becomes active. A constructor is also invoked when a temporary object of the class is created.

```
class X {  
public:  
    X();    // class X constructor  
};
```

- ◆ A class X constructor cannot take X as an argument:

```
class X {  
public:  
    X(X);    // illegal  
};
```

- ◆ The parameters to the constructor can be of any type except that of the class it's a member of. The constructor can accept a reference to its own class as a parameter; when it does so, it is called the copy constructor . A constructor that accepts no parameters is called the default constructor.

C++ - Constructors

(cont)

- ◆ There are two ways to initialise the data members with the constructor:

```
class employee {
private:
    String name;    // Embedded class
    int staff_id;
    float salary;
public:
    employee (const String& n, int id, float sal);
    ...
};
```

C++ - Constructors

(cont)

- ◆ The first way is to use the member initialisation list, e.g:

```
employee::employee (const String& n, int id, float sal)
    :      name(n), staff_id(id), salary(sal) { }
```

- ◆ The second way to initialise members is to assign to them within the body of the constructor, e.g.:

```
employee::employee (const String& n, int id, float sal)
{
    name = n;
    staff_id = id;
    salary = sal;
}
```

C++ - Destructors

- ◆ The destructor for a class is called to free members of an object before the object is itself destroyed. The destructor is a member function whose name is that of the class preceded by a tilde (~). A destructor cannot accept any parameters, nor will it have a return type or value declared.

```
#include <stdlib.h>
class X
{
public:
    ~X(){}; // destructor for class X
};
```

- ◆ If a destructor isn't explicitly defined for a class, the compiler generates one (see example condes.cxx).

C++ - Friend Functions

◆ Syntax

```
friend <identifier>;
```

◆ Description

- Use friend to declare a function with full access rights to the private and protected members of an outside class, without being a member of that class.
- In all other respects, the friend is a normal function in terms of scope, declarations, and definitions.

(See example friend.cxx)

C++ - Operators

◆ Syntax

```
operator <operator symbol>( <parameters> )  
{  
    <statements>;  
}
```

◆ Description

- Use the operator keyword to define a new (overloaded) action of the given operator. When the operator is overloaded as a member function, only one argument is allowed, as `*this` is implicitly the first argument.
- When you overload an operator as a friend, you can specify two arguments (see example `operators.cxx`).

- ◆ You cannot change the number of operands for an overloaded operator. Nor can the order of operands be changed for asymmetric operators (e.g. `->`).
- ◆ There are two ways to implement overloaded operators: as member functions or as global non-member functions (usually friend functions).
- ◆ Apart from `operator=()` (assignment), all member function operators can be inherited (see section 2 for a discussion of the assignment operator). All member function operators can be made virtual.
- ◆ Friend operators cannot be inherited or made virtual (because they are not part of the class).

C++ - Operators

(cont)

- ◆ Overloaded operators cannot have default arguments.
- ◆ Define operators, where possible, in terms of other overloaded functions, so as to minimise future changes (e.g. if operator == has been defined so that expressions such as a==b can be made, define operator != as the negation of operator ==, i.e. !(a == b))
- ◆ operator=, operator(), operator-> and operator[] must be implemented as member functions.
- ◆ No user-defined conversions will be invoked on the first operand (this) of an expression which calls an overloaded operator defined as a member function.
- ◆ In general, all symmetric operators (arithmetic and relational) should be defined as friends to enable conversions to be applied to either operand. All asymmetric operators ((), [], unary * and ->) must be defined as members, thereby enforcing that the left-hand side operand is an lvalue.

C++ - Inheritance

- ◆ Inheritance is a relationship between classes whereby one class (called derived class) acquires the structure (I.e. data member and member functions) of other classes (called base classes) in a strict hierarchy from either a single parent (single inheritance) or multiple (multiple inheritance) parents (see example `ac_cnt.cxx`).

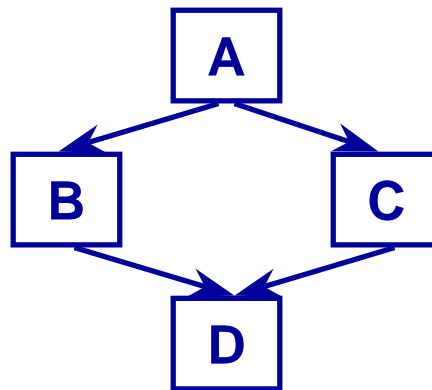
C++ - Inheritance

(cont)

- ◆ A derived class may have different access rights to its base class(es), and namely (see exercise 3):

	Type of Inheritance		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

- ◆ When a derived class is created the constructor(s) of its base class(es) are also called. To **select the proper constructors** to be called the **initialisation list mechanism** can be used (see example `c2cpp2.cxx`).
- ◆ When multiple inheritance is used, care must be taken not to introduce **cycles in the hierarchy graphs** e.g.:



C++ - Non Virtual / Virtual / Pure Virtual

◆ Syntax

```
virtual class-name  
virtual function-name
```

◆ Description

- Use the virtual keyword to allow derived classes to provide different versions of a base class function. Once you declare a function as virtual, you can redefine it in any derived class, even if the number and type of arguments are the same.
 - The redefined function overrides the base class function.
- ◆ Let's have a look to the examples `c2cpp2.cxx`, `c2cpp3.cxx`, `c2cpp4.cxx` and check how the virtual mechanism works.

C++ - Non Virtual / Virtual / Pure Virtual (c)

- ◆ Notice that there are two components to public inheritance: inheritance of function interfaces (declarations) and of function implementations (definitions). Member function interfaces are always inherited in public inheritance. Whether an implementation is inherited depends on whether a function is non-virtual, virtual or pure virtual.
- ◆ If a function is non-virtual in a base class, derived classes are bound to use that implementation.
- ◆ If a function is (simple) virtual in a base class, derived classes inherit a default implementation from their base class. If desired, this default implementation can be overridden if the derived class redefines (and reimplements) the virtual function.

C++ - Non Virtual / Virtual / Pure Virtual (c)

- ◆ If a function is pure virtual, it allows derived classes to inherit the interface - no implementation is inherited. The derived class must provide an implementation.

C++ - Memory Allocation Problems

- ◆ C++ inherits all the memory allocation problems of C.
- ◆ On top of that C++ introduces its own memory allocation problems, and namely:
 - problems occurring when an object is assigned to another one;
 - problems occurring when an object is created;
 - problems caused by inheritance.

C++ - Memory Allocation Problems - Constructors

(cont)

- ◆ If a class (or any of its base classes) contains pointer members then there is a strong likelihood that a copy constructor is required. The key issue is:
- ◆ "Is there any difference between a) copying an object and b) copying its data members and base classes?"
- ◆ If the answer is "yes", then a copy constructor is needed.
- ◆ In the absence of a copy constructor, the compiler will generate a shallow copy implementation to perform a bitwise copy of the original. In trivial classes this is sufficient, but consider the following example.

C++ - Memory Allocation Problems - Constructors

(cont)

```
class String {
public:
    String (char*);
    void print () { cout << string_rep; }
private:
    char* string_rep;
};

main ()
{
    String *A = new String ("A new string\n");
    String B = *A;

    delete A;
    .....
    B.print();           // Result not guaranteed !
}
```

C++ - Memory Allocation Problems - Constructors

(cont)

- ◆ The line `String B = *A;` will cause the compiler to generate a shallow copy constructor. As a result, only the `string_rep` pointer is copied to `B`, not what it actually points to. Therefore, when `A` is deleted, so is `string_rep`; since `B`'s `string_rep` is the same thing, `B`'s `string_rep` is invalidated. In a simple application, the problem may go unnoticed because the memory has not been reused. However, very obscure bugs may appear in larger programs.
- ◆ To avoid such problems, the `String` class author must provide a copy constructor to perform a deep copy, as shown here below.

C++ - Memory Allocation Problems - Constructors

(cont)

```
class String {
public:
    String (char*);
    String (const String&);
    void print () { cout << string_rep; }
private:
    char* string_rep;
};

String::String (const String& s2)
{
    string_rep = new char [::strlen(s2)+1];
    ::strcpy (string_rep, s2.string_rep);
}
```

C++ - Memory Allocation Problems - Constructors

(cont)

- ◆ Note: if a class author does not wish objects of a class to be copied, it is not sufficient to simply not provide a copy constructor because the compiler will generate one. Instead, the author must make the copy constructor private. No implementation need to be provided. When a programmer attempts to copy objects with private copy constructors and assignment operators, the compiler will generate an error.

C++ - Memory Allocation Problems - Assignment

(cont)

- ◆ The need for an assignment operator is closely related to the need for a copy constructor. As with the copy constructor, if there is no assignment operator, the compiler will generate a shallow copy (bitwise) version. If the class or its base classes contain pointer members then a shallow copy is insufficient (for the reasons explained above for copy constructors).
- ◆ Therefore, if a copy constructor is needed, it is almost certain that an assignment operator will be needed. The actual copying process is similar to that for the corresponding copy constructor, apart from one important addition: the overloaded assignment operator must check for the special case of assignment of an object to itself (i.e. $a=a$).
- ◆ The String class used above would implement the following assignment operator:

C++ - Memory Allocation Problems - Assignment

(cont)

```
class String {
public:
    String (char*);
    String (const String&);
    String& operator=(const String&);
    void print () { cout << string_rep; }
private:
    char* string_rep;
};

String& String::operator = (const String& s2)
{
    if (this != &s2)
    {
        if (string_rep)
        {
            delete [] string_rep;
        }
        string_rep = new char [::strlen(s2)+1];
        ::strcpy (string_rep, s2.string_rep);
    }
    return *this;
}
```

C++ - Memory Allocation Problems - Assignment (cont)

- ◆ The assignment operator must first check for self-assignment. If this and `&s2` are different, then the assignment is not a self-assignment. If the `string_rep` member had been set up it is deleted. The `string_rep` member is then reinitialised (by calling `new`) and the new value copied in. Finally the current object is returned as the result.
- ◆ It is now clear why the self-assignment test is necessary. Without it, `string_rep` is deallocated before it is copied. The results of this operation are not defined by C++.
- ◆ Note that, as is the case for copy constructors, if the class author wishes to prevent assignment, the assignment operator should be made private (no implementation need be provided).

C++ - Memory Allocation Problems - Inheritance

(cont)

- ◆ If a class is intended to be used (now or in the future) as a base class, it is imperative that its destructor be made virtual. Some classes may not need destructors because they are so trivial. Other classes must have destructors to perform tidy-up operations, e.g. file closing, memory deallocation, etc.
- ◆ Some types of trivial classes are often used as (abstract) base classes - they define virtual functions, often with no implementation (pure virtual) and serve to provide an interface to a particular class hierarchy. Even though they may be so trivial as to not require a destructor, such base classes must specify a virtual destructor to enable correct destruction of derived objects. If the base class would otherwise have no need for a destructor, then an empty implementation should be provided.
- ◆ Why must the destructor be virtual? Consider the following example:

C++ - Memory Allocation Problems - Inheritance

(cont)

```
class shape {
    virtual void draw() = 0;           // Pure virtual
    virtual void erase() = 0;         // member functions.
    ....
public:
    shape() { }
    ~shape() { }                       //
};

class square : public shape {
private:
    char* title;
    int x1, y1, x2, y2;
public:
    square ();
    square (int a,int b,int c,int d,char*=0);
    ~square ();
    void draw ();
    void erase ();
    ...
};
```

C++ - Memory Allocation Problems - Inheritance

(cont)

```
void display_a_shape (shape* s)
{
    s->draw();
}

main ()
{
    square* sqr1 = new square(1,1,10,10);
    shape* sqr2 = new square(2,2,9,9);

    display_a_shape (sqr1);
    display_a_shape (sqr2);

    delete sqr1;          // ok - calls square::~~square()
    delete sqr2;          // wrong - calls shape::~~shape()
}
```

C++ - Memory Allocation Problems - Inheritance

(cont)

- ◆ The problem when deleting `sqr2` is that, because the `shape` destructor is not virtual, it is itself called rather than the `square` destructor. This leaves the memory allocated for the `square` object's data members "dangling", unable to be freed until program termination. If this process were repeated many times it would gradually consume more and more of the free store.
- ◆ So far we have assumed that the base class does not itself allocate a system resource. If it does, then all derived class constructors must necessarily call the base class constructor in the member initialisation list (or else rely on implicitly invoking the base class's default constructor). Ensuring that the base class destructor is virtual means that tidy-up of the derived class is performed, after which C++ automatically invokes the destructors for any base classes.

C++ - Memory Allocation Problems - Inheritance

(cont)

- ◆ What then are the rules for destructors?
 - **If a program dynamically allocates** some system resource (e.g. files, memory, etc) then a **destructor should be implemented**.
 - **If any member functions are virtual**, then this is a clear statement that you intend a class to be inherited. In this case, **a virtual destructor must be provided**, even if the class would otherwise not need a destructor.

C++ - “Orthodox Canonical” Class Format

```
class stock_item {
friend x,y;
public:
    // 1) Provide a default constructor
    stock_item ();
    // 2) Make sure that all members are initialised:
    stock_item (int sn, char* title);
    // 3) Provide accessors:
    int get_serial_number() const;
    ....
    // 4) Provide modifiers:
    void set_serial_number(int sn);
    //=====
    // 5) Either provide a non-virtual destructor if no
    //      member functions are virtual...
    ~stock_item ();
```

C++ - “Orthodox Canonical” Class Format (c)

```
//=====
// 5) Either provide a non-virtual destructor if no
//      member functions are virtual...
~stock_item ();
// 5) ...or provide a virtual destructor if any
//      member functions are virtual:
virtual void print ();
virtual ~stock_item ();
//=====
// 6) Provide a copy constructor:
stock_item (const stock_item&);
// 7) Provide an assignment operator which
//      correctly handles assignment to itself:
stock_item& operator = (const stock_item&);
private:
// 8) Make sure that no data members are public:
int serial_number;
char* name;
};
```

C++ - Templates (I.e. Parametric Polymorphism) - STDM

- ◆ Templates are a way of defining “parametric” (“generic”) classes and/or functions (I.e. generic pieces of code able to work on objects whose types have not been yet specified).
- ◆ Examples of **class templates** are given in the program `c2cpp5.cxx`.
- ◆ An example of **function templates** is given in the book “The C++ Programming Language” on page 334.
- ◆ STL (the Standard Templates Library) has been adopted as part of the C++ language and few free implementations are available; their usage should be encouraged (in ground SW applications).

C++ - Exception Handling - STDM

- ◆ The C++ language defines a standard for exception handling. The standard insures that the power of object-oriented design is supported throughout your program. An especially strong feature of the standard is the availability of virtual functions and the use of objects to define exceptions. Virtual functions guarantee a minimum of runtime overhead--zero additional program overhead if no exceptions are thrown.
- ◆ When an abnormal situation arises at runtime, the program should terminate. However, throwing an exception allows you to gather information at the throw point that could be useful in diagnosing the causes which led to failure. You can also specify in the exception handler the actions to be taken before the program terminates. Only synchronous exceptions are handled, meaning that the cause of failure is generated from within the program. An event such as Control-C (which is generated from outside the program) is not considered to be an exception.
(See example ioexp.cxx)

C++ - Exception Handling -

(cont)

- ◆ When the program encounters an abnormal situation for which it is not designed, you may transfer control to some other part of the program that is designed to deal with the problem. This is done by throwing an exception.
- ◆ The exception-handling mechanism requires the use of three keywords: try, catch, and throw. The try-block specified by try must be followed immediately by the handler specified by catch. If an exception is thrown in the try-block, program control is transferred to the appropriate exception handler. The program should attempt to catch any exception that is thrown by any function. Failure to do so could result in abnormal termination of program.
- ◆ Although C++ allows an exception to be of any type, it is useful to make exceptions objects. The exception object is treated exactly the way any object would. An exception carries information from the point where the exception is thrown to the point where the exception is caught. This is information that the program user will want to know when the program encounters some anomaly at runtime.

- ◆ A block of code in which an exception can occur must be prefixed by the keyword **try**. Following the **try** keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the program flow is interrupted. The sequence of steps taken is as follows:
 - The program searches for a matching handler
 - If a handler is found, the stack is unwound to that point; program control is transferred to the handler
 - If no handler is found, the program will call the `terminate` function. If no exceptions are thrown, the program executes in the normal fashion.

C++ - Exceptions - catch

(cont)

- ◆ The exception handler is indicated by the **catch** keyword. The handler must be used immediately after the try-block. The keyword catch can also occur immediately after another catch
- ◆ Each handler will only evaluate an exception that matches, or can be converted to, the type specified in its argument list.
- ◆ Every exception thrown by the program must be caught and processed by the exception handler. If the program fails to provide an exception handler for a thrown exception, the program will call terminate.
- ◆ Exception handlers are evaluated in the order that they are encountered. An exception is caught when its type matches the type in the catch statement. Once a type match is made, program control is transferred to the handler. The stack will have been unwound upon entering the handler. The handler specifies what actions should be taken to deal with the program anomaly.

C++ - Exceptions - catch

(cont)

- ◆ A goto statement can be used to transfer program control out of a handler but such a statement can never be used to enter a handler.
- ◆ After the handler has executed, the program can continue at the point after the last handler for the current try-block. No other handlers are evaluated for the current exception.

C++ - Exceptions - catch

(cont)

◆ Example 1

```
try {  
    // Include any code that might throw an exception  
}  
catch (T X) // Provide a handler for each exception that might be  
thrown above  
{  
    // Take some actions  
}
```

- ◆ This example is specifically defined to handle an object of type T. If the argument is T, T&, const T, or const T&, the handler will accept an object of type X if any of the following are true:
 - T and X are of the same type
 - T is an accessible base class for X in the throw expression
 - T is a pointer type and X is a pointer type that can be converted to T by a standard pointer conversion in the throw expression

C++ - Exceptions - catch

(cont)

- ◆ Example 2

```
try {  
    // Include any code that might throw an exception  
}  
catch ( ... )  
{  
    // Take some actions  
}
```

- ◆ The statement `catch (...)` will handle any exception, regardless of type. This statement, if used, must be the last handler for its try-block.

- ◆ A throw expression is also referred to as a throw-point. You can specify whether an exception may be thrown by using one of the following syntax specifications:
- ◆ When an exception occurs, the throw expression initializes a temporary object of the type T (to match the type of argument arg) used in `throw(T arg)`. Other copies can be generated as required by the compiler. Consequently, it can be useful to define a copy constructor for the exception object.

- ◆ Example 1

```
throw throw_object;
```

This example specifies that `throw_object` is to be passed to a handler.

- ◆ Example 2

```
throw;
```

This example simply specifies that the last exception thrown is to be thrown again. An exception must currently exist. Otherwise, `terminate` is called.

C++ - Exceptions - throw

(cont)

- ◆ Example 3

```
void my_func1() throw (A, B)
{
    // Body of function.
}
```

This example specifies a list of exceptions that `my_func1` can throw. No other exceptions will propagate out of `my_func1`. If an exception other than A or B is generated within `my_func1`, it is considered to be an unexpected exception and program control will be transferred to the unexpected function.

- ◆ Example 4

```
void my_func2() throw ()
{
    // Body of this function.
}
```

The final case specifies that `my_func2` will throw no exceptions. If any function in the body of `my_func2` throws an exception, such an exception will not exist beyond the body of `my_func2`.

C++ - Exceptions - Specification

(cont)

- ◆ The C++ language makes it possible for you to specify any exceptions that a function can throw. This exception specification can be used as a suffix to the function declaration. The syntax for exception specification is as follows:

```
exception-specification:
```

```
    throw (type-id-listopt)
```

```
type-id-list:
```

```
    type-id
```

```
    type-id-list, type-id
```

- ◆ The function suffix is not considered to be part of the function's type. Consequently, a pointer to a function is not affected by the function's exception specification. Such a pointer checks only the function's return and argument types. Therefore, the following is legal:

C++ - Exceptions - Specification (cont)

```
void f2(void) throw();           // Should not throw
exceptions
void f3(void) throw (BETA);     // Should only
throw BETA objects
void (* fptr)();                // Pointer to a
function returning void
fptr = f2;
fptr = f3;
```

- ◆ Extreme care should be taken when overriding virtual functions. Again, because the exception specification is not considered part of the function type, it is possible to violate the program design.
- ◆ If an exception is thrown which is not listed in the exception specification, the unexpected function will be called.

C++ - Exceptions - unexpected

(cont)

◆ Syntax

```
void unexpected();
```

◆ Description

- The `unexpected` function is called when a function throws an exception not listed in its exception specification. The program calls `unexpected`, which by default calls any user-defined function registered by `set_unexpected`. If no function is registered with `set_unexpected`, the `unexpected` function then calls `terminate`.

◆ Return Value

- None, although `unexpected` may throw an exception.

◆ Syntax

```
void terminate();
```

◆ Description

- The function `terminate` can be called by unexpected or by the program when a handler for an exception cannot be found. The default action by `terminate` is to call `abort`. Such a default action causes immediate program termination.
- You can modify the way that your program will terminate when an exception is generated that is not listed in the exception specification. If you do not want the program to terminate with a call to `abort`, you can instead define a function to be called. Such a function (called a `terminate_function`) will be called by `terminate` if it is registered with `set_terminate`.

C++ - Namespaces - STDM

- ◆ The namespace mechanism allows an application to be partitioned into number of subsystems. Each subsystem can define and operate within its own scope. Each developer is free to introduce whatever identifiers are convenient within a subsystem without worrying about whether such identifiers are being used by someone else. The subsystem scope is known throughout the application by a unique identifier.
- ◆ It only takes two steps to use C++ namespaces. The first is to uniquely identify a name space with the keyword **namespace**. The second is to access the elements of an identified namespace by applying the **using** keyword (See example namespaces.cxx).

C++ - Namespaces

(cont)

```
// A namespace example
#include <iostream.h>
    namespace F {
        float x = 9;
    }
    namespace G {
        using namespace F;
        float y = 2.0;
        namespace INNER_G {
            float z = 10.01;
        }
    }
    ./.

```

C++ - Namespaces

(cont)

```
./ int main() {  
  
    using namespace G;  
    using namespace G::INNER_G;  
    float x = 19.1;  
  
    cout << "x = " << x << endl;  
    cout << "y = " << y << endl;  
    cout << "z = " << z << endl;  
  
    return 0;  
}
```

C++ - Casts - STDm

◆ Syntax

`cast_type<type-id>(expression)`

◆ Description

- To improve type safety the old cast construct has now 4 different versions:
 - » **const_cast**: used to remove the const, volatile, and `__unaligned` attribute(s) from a class;
 - » **static_cast**: it converts expression to the type of type-id based solely on the types present in the expression;
 - » **reinterpret_cast**: it allows any pointer to be converted into any other pointer type, and it allows any integral type to be converted into any pointer type and vice versa;
 - » **dynamic_cast**: it converts the operand expression to an object of type type-id; it used to navigate along class hierarchies.

C++ - Const Cast - STDm

◆ Syntax

```
const_cast <type-id> (expression)
```

◆ Description

- A pointer to any object type or a pointer to a data member can be explicitly converted to a type that is identical except for the `const`, `volatile`, and `__unaligned` qualifiers. For pointers and references, the result will refer to the original object. For pointers to data members, the result will refer to the same member as the original (uncast) pointer to data member.
- The `const_cast` operator converts a null pointer value to the null pointer value of the destination type.

C++ - Static Cast - STDm

◆ Syntax

```
static_cast <type-id> (expression)
```

◆ Description

- The expression `static_cast < type-id > (expression)` converts expression to the type of `type-id` based solely on the types present in the expression. No run-time type check is made to ensure the safety of the conversion.
- The `static_cast` operator can be used for operations such as converting a pointer to a base class to a pointer to a derived class. Such conversions are not always safe.

C++ - Reinterpret Cast - STDm

◆ Syntax

```
reinterpret_cast <type-id> (expression)
```

◆ Description

- The `reinterpret_cast` operator allows any pointer to be converted into any other pointer type. It also allows any integral type to be converted into any pointer type and vice versa. Misuse of the `reinterpret_cast` operator can easily be unsafe.

C++ - Dynamic Cast - STDM

◆ Syntax

```
dynamic_cast <type-id> (expression)
```

◆ Description

- The expression `dynamic_cast<type-id>(expression)` converts the operand expression to an object of type `type-id`. The `type-id` must be a pointer or a reference to a previously defined class type or a "pointer to void". The type of expression must be a pointer if `type-id` is a pointer, or an l-value if `type-id` is a reference.
- If `type-id` is a pointer to an unambiguous accessible direct or indirect base class of expression, a pointer to the unique subobject of type `type-id` is the result (upcast).
- If the type of expression is a base class of the type of `type-id`, a run-time check is made to see if expression actually points to a complete object of the type of `type-id`. If this is true, the result is a pointer to a complete object of the type of `type-id` (downcast).

C++ - RTTI - STDM

- ◆ Run time type information (RTTI) is a mechanism that allows the type of an object to be determined during program execution.
- ◆ There are three main C++ language elements to run-time type information (see example `tinfo.cxx`):
 - the **dynamic_cast** operator: used for conversion of polymorphic types;
 - the **typeid** operator, used for identifying the exact type of an object.
 - the **type_info** class, used to hold the type information returned by the `typeid` operator.

C++ - Summary

- ◆ “A Better C”
- ◆ The Concept of Class
- ◆ Access Control
- ◆ Static Members
- ◆ this
- ◆ Inline
- ◆ Constructors
- ◆ Destructors
- ◆ Friend Functions
- ◆ Operators
- ◆ Inheritance
- ◆ Virtual / Non Virtual / Pure Virtual
- ◆ Memory Allocation Problems
- ◆ “Orthodox Canonical” Class Format
- ◆ Templates
- ◆ Exceptions
- ◆ Namespaces
- ◆ Casts
- ◆ RTTI

C++ - Exercises

- ◆ 1. Modify the programs `c2cpp0.c` and `c2cpp1.c` so that a new type `Circle` is introduced (with data `x`, `y`, `radius` and function `display_circle`). Comment the results.
- ◆ 2. Modify the original version of the program `c2cpp1.c` so that the pointer to the function `void (*display)(void * arg)` goes at the end of the struct definitions. Comment the results.
- ◆ 3. Change the keyword **public** in line 12 of the example `ac_cnt.cxx` to **protected** first and **private** later and comment the results.
- ◆ 4. Modify the example program `c2cpp4.cxx` so that the class `GraphicObject` is embedded (I.e. included) into `Line`, `Rectangle` and `Triangle`. Comment the differences between the two implementations.
- ◆ 5. Use the example on page 334 of “The C++ Programming Language” to rewrite in C++ the C exercise n. 8 (a sort program).

- ◆ 6. Study carefully the File member functions in the example `ioexp.cxx` and try to understand what they do.
- ◆ 7. Read the section on Exceptions in chapter 8 of “The C++ Programming Language” and compare it with the example `exceptio.cxx`. Comment the differences between the expected and actual behaviour.
- ◆ 8. Give some examples where the application of the RTTI mechanism is beneficial.

C++ - Further Reading

- ◆ Bjarne Stroustrup, “The C++ Programming language, 3rd Edition, Addison Wesley, 1997, ISBN 0-201-88954-4.
- ◆ <http://www.ocsltd.com/c++/index.html>, to get information on the C++ standard and all its major and minor extensions/modification to the language.