Welcome to C++ is fun at Turbine/Warner Bros.!

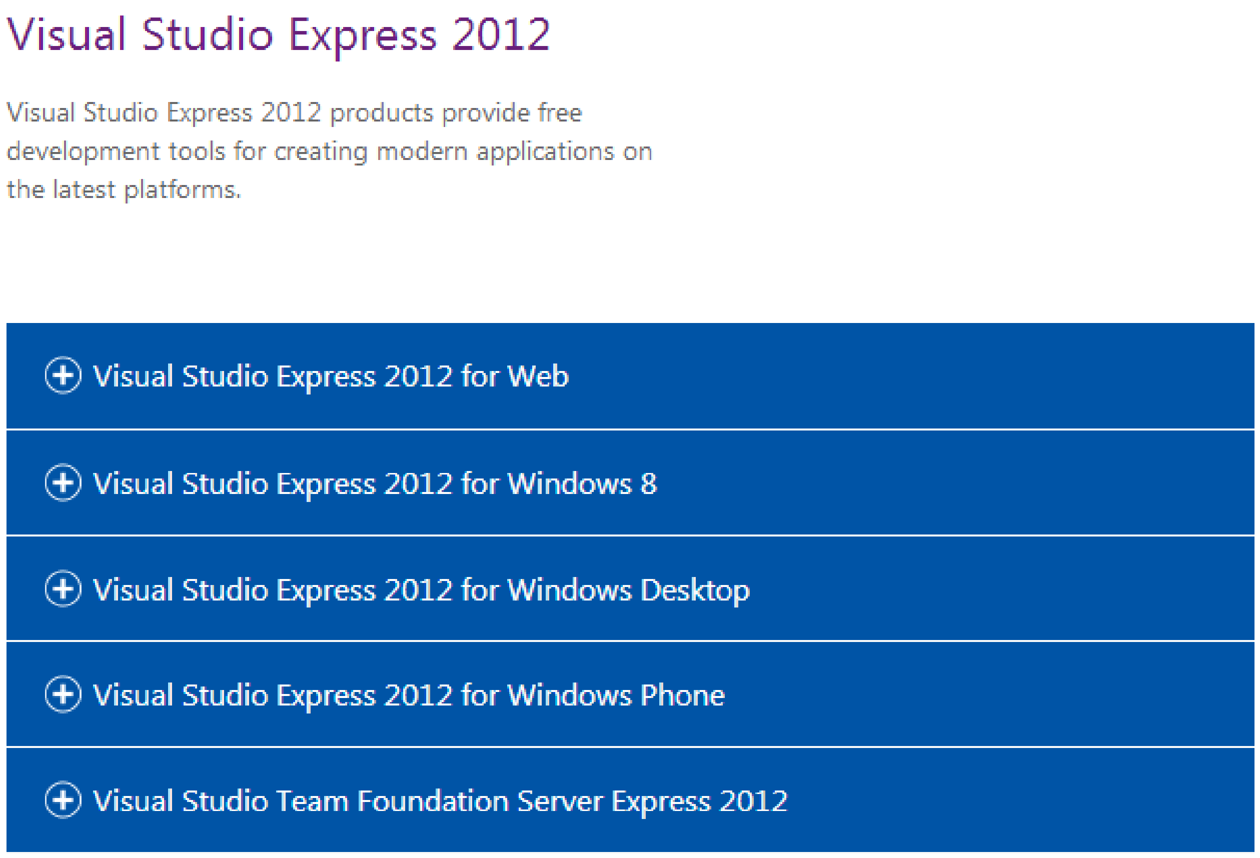
Russell Hanson

Syllabus

1) First program and introduction to data types and control structures with applications for games learning how to use the programming environment
2) Objects, encapsulation, abstract data types, data protection and scope
3) Basic data structures and how to use them, opening files and performing operations on files
<u>Mini Project 1 Due</u>
4) Algorithms on data structures, algorithms for specific tasks, simple AI and planning type algorithms, game AI algorithms
6) Event-oriented programming, model view controller, map reduce filter
<u>Mini Project 2 Due</u>
7) Basic threads models and some simple databases SQLlite
8) Graphics programming, shaders, textures, 3D models and rotations
7) How to download an API and learn how to use functions in that API, Windows Foundation Classes
8) Designing and implementing a simple game in C++
9) Selected topics - depth controllers like the Microsoft Kinect
<u>Mini Project 3 Due</u>
10) Selected topics
11) Working on student projects
12) Final project presentations

Visual Studio 2012

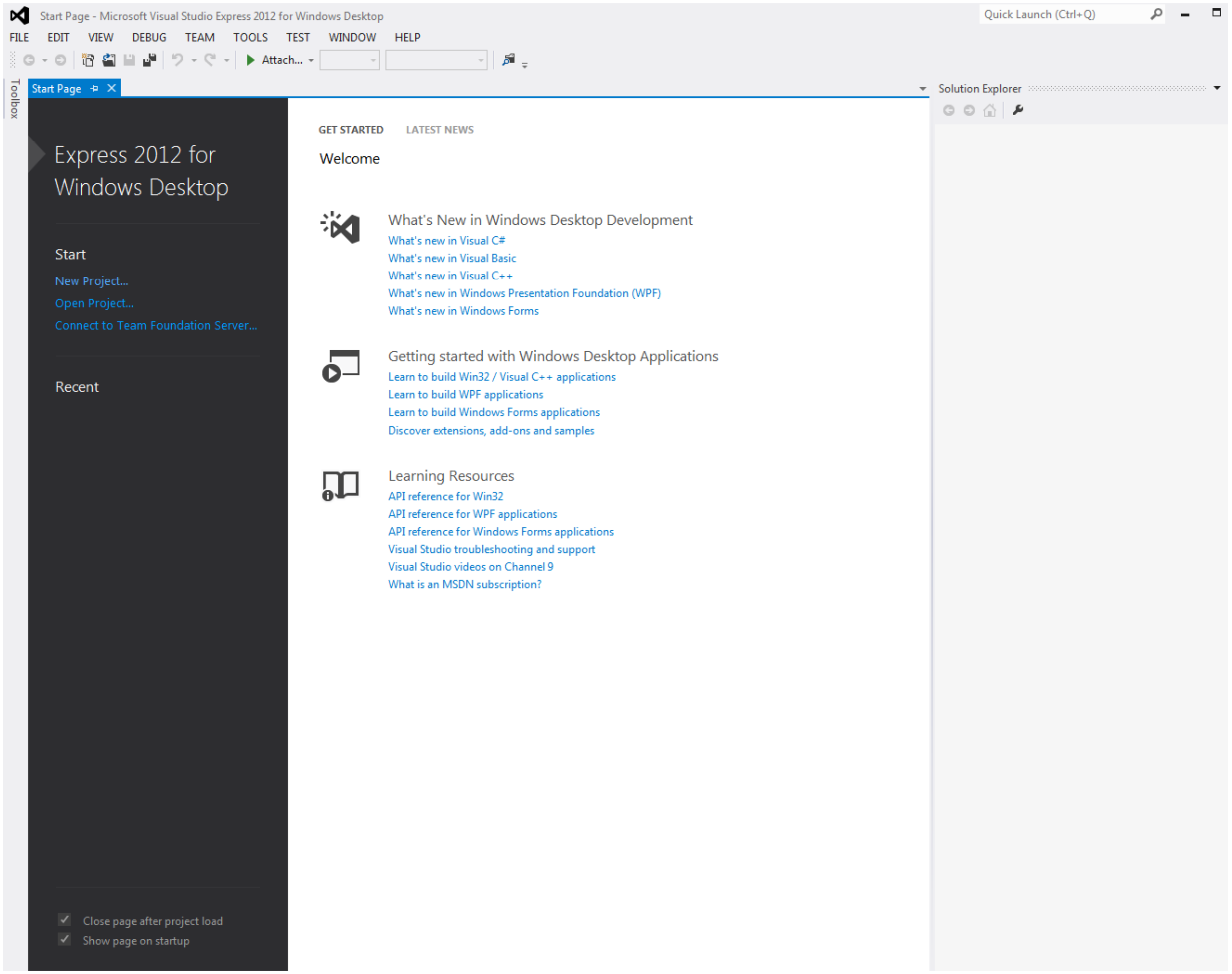
http://www.microsoft.com/visualstudio/eng/downloads



Visual Studio Express 2012

Visual Studio Express 2012 products provide free development tools for creating modern applications on the latest platforms.

Visual Studio Express 2012 for Web
Visual Studio Express 2012 for Windows 8
Visual Studio Express 2012 for Windows Desktop
Visual Studio Express 2012 for Windows Phone
Visual Studio Team Foundation Server Express 2012



Express 2012 for Windows Desktop

Start

New Project...

Open Project...

Connect to Team Foundation Server...

Recent

☑ Close page after project load

☑ Show page on startup

GET STARTED LATEST NEWS

Welcome

What's New in Windows Desktop Development

What's new in Visual C#

What's new in Visual Basic

What's new in Visual C++

What's new in Windows Presentation Foundation (WPF)

What's new in Windows Forms

Getting started with Windows Desktop Applications

Learn to build Win32 / Visual C++ applications

Learn to build WPF applications

Learn to build Windows Forms applications

Discover extensions, add-ons and samples

Learning Resources

API reference for Win32

API reference for WPF applications

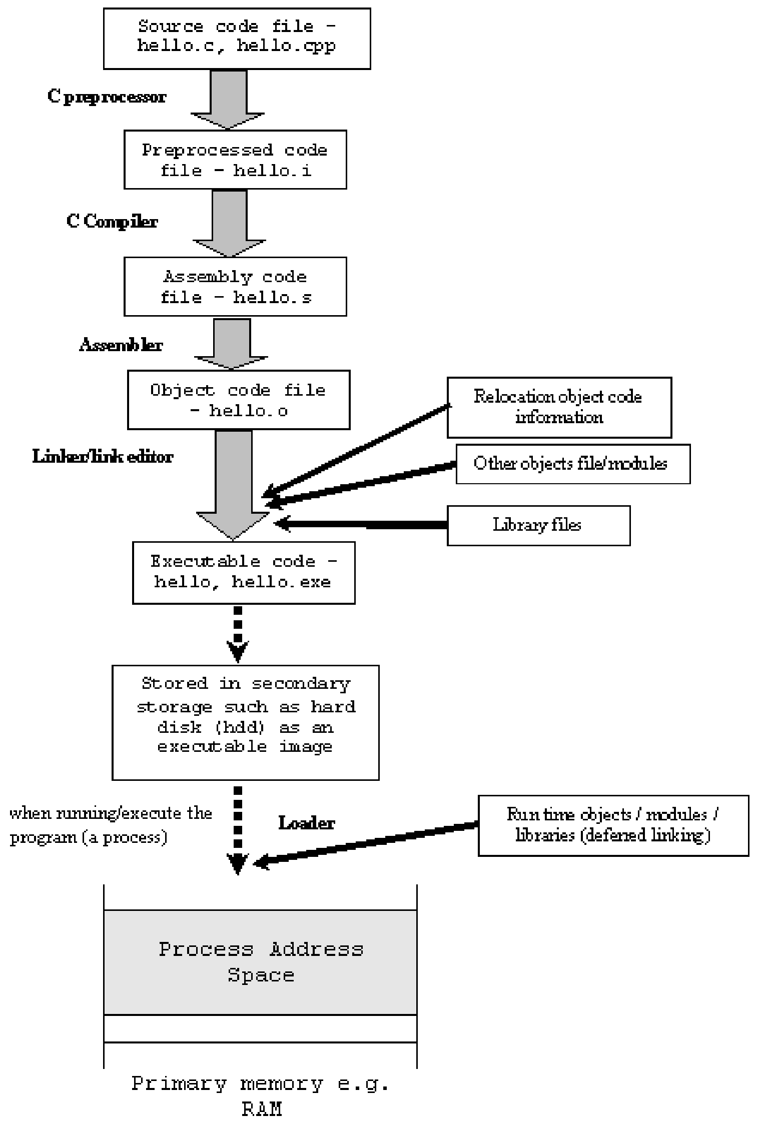
API reference for Windows Forms applications

Visual Studio troubleshooting and support

Visual Studio videos on Channel 9

What is an MSDN subscription?

What happens when you write a program?

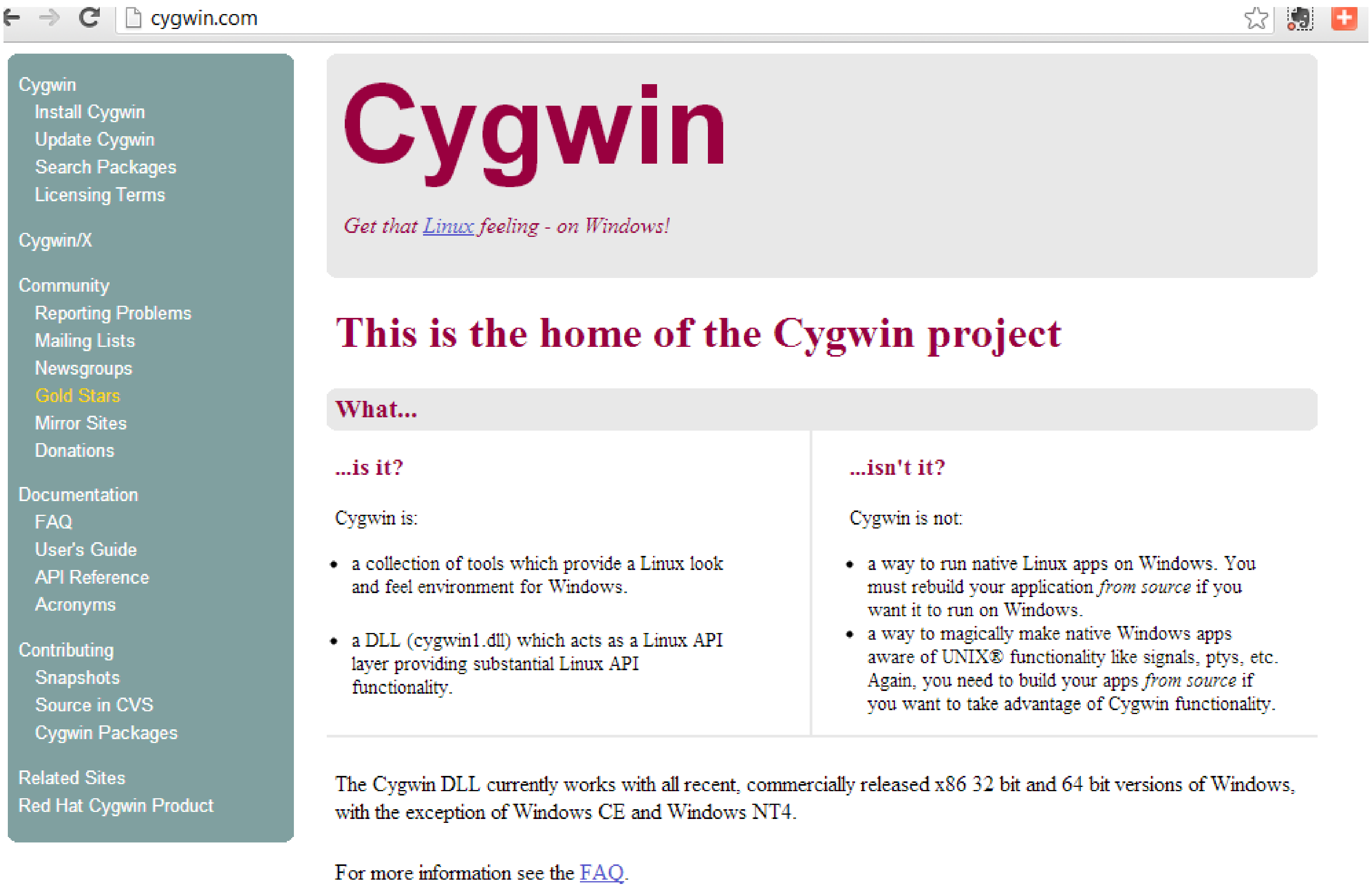


Source code file - hello.c, hello.cpp
C preprocessor
Preprocessed code file - hello.i
C Compiler
Assembly code file - hello.s
Assembler
Object code file - hello.o
Linker/link editor
Relocation object code information
Other objects file/modules
Library files
Executable code - hello, hello.exe
Stored in secondary storage such as hard disk (hdd) as an executable image
when running/execute the program (a process)
Loader
Run time objects / modules / libraries (deferred linking)
Process Address Space
Primary memory e.g. RAM

More reading:

http://www.tenouk.com/ModuleW.html

More Linux/Unix style from the command prompt



Cygwin
- Install Cygwin
- Update Cygwin
- Search Packages
- Licensing Terms

Cygwin/X

Community
- Reporting Problems
- Mailing Lists
- Newsgroups
- Gold Stars
- Mirror Sites
- Donations

Documentation
- FAQ
- User's Guide
- API Reference
- Acronyms

Contributing
- Snapshots
- Source in CVS
- Cygwin Packages

Related Sites
Red Hat Cygwin Product

Cygwin

Get that Linux feeling - on Windows!

This is the home of the Cygwin project

What...

...is it?

Cygwin is:

- a collection of tools which provide a Linux look and feel environment for Windows.
- a DLL (cygwin1.dll) which acts as a Linux API layer providing substantial Linux API functionality.

...isn't it?

Cygwin is not:

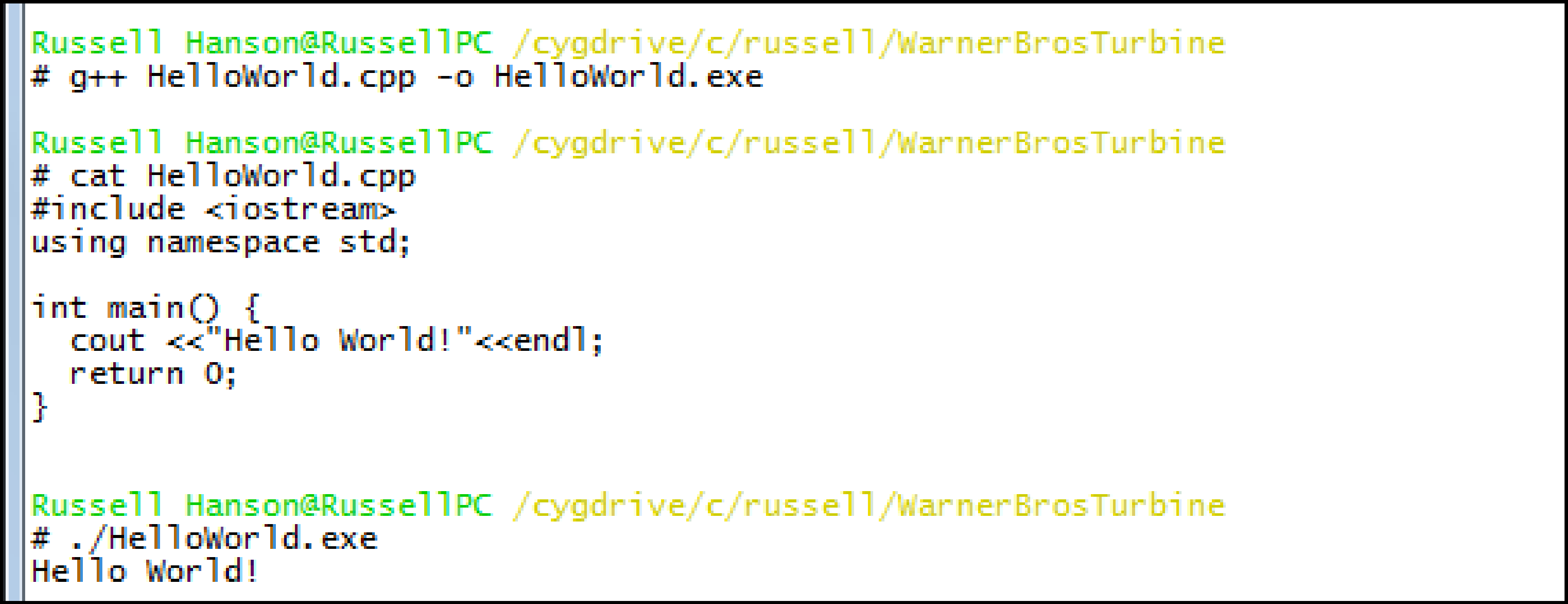
- a way to run native Linux apps on Windows. You must rebuild your application *from source* if you want it to run on Windows.
- a way to magically make native Windows apps aware of UNIX® functionality like signals, ptys, etc. Again, you need to build your apps *from source* if you want to take advantage of Cygwin functionality.

The Cygwin DLL currently works with all recent, commercially released x86 32 bit and 64 bit versions of Windows, with the exception of Windows CE and Windows NT4.

For more information see the FAQ.

First Program, “Hello World!”

```cpp
#include <iostream>
using namespace std;
int main() {
cout <<"Hello World!"<<endl;
return 0;
}
```



```
Russell Hanson@RussellPC /cygdrive/c/russell/WarnerBrosTurbine
# g++ HelloWorld.cpp -o HelloWorld.exe

Russell Hanson@RussellPC /cygdrive/c/russell/WarnerBrosTurbine
# cat HelloWorld.cpp
#include <iostream>
using namespace std;

int main() {
  cout <<"Hello World!"<<endl;
  return 0;
}


Russell Hanson@RussellPC /cygdrive/c/russell/WarnerBrosTurbine
# ./HelloWorld.exe
Hello World!
```

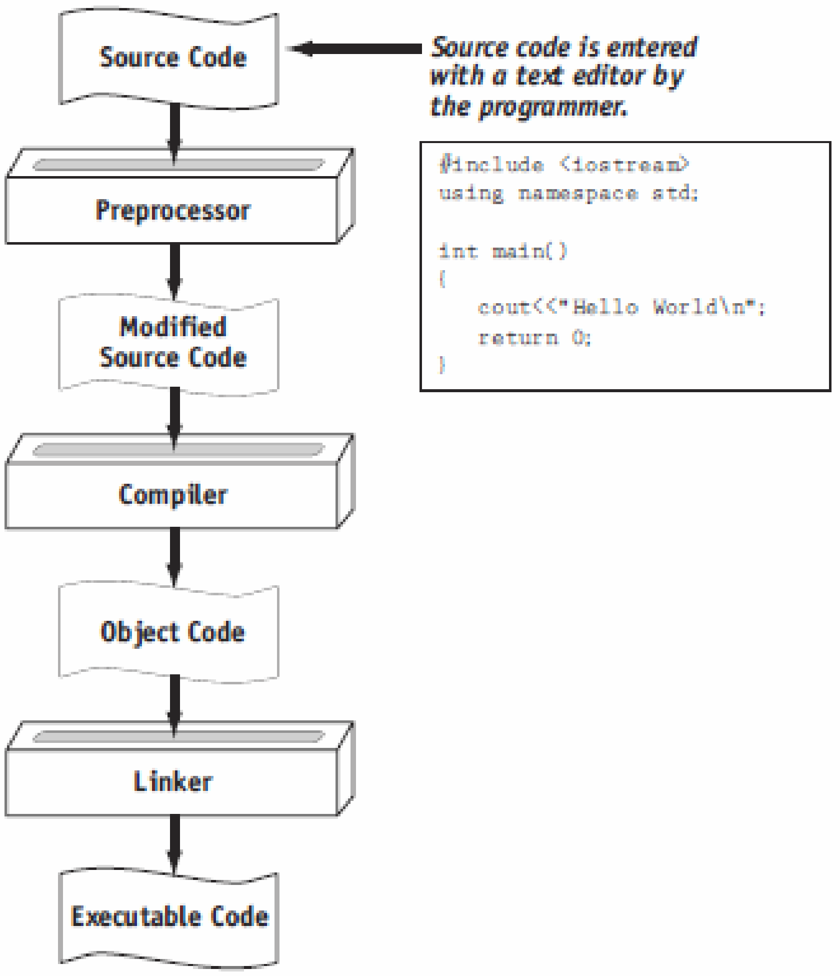
```c
#include <stdafx.h>
#include <stdio.h>
static void display(int i, int *ptr);

int main(void)
{
	int x = 5;
	int *xptr = &x;
	printf("In main() program:\n");
	printf("x value is %d and is stored at address %p.\n", x, &x);
	printf("xptr pointer points to address %p which holds a value of %d.\n", xptr, *xptr);
	display(x, xptr);
	return 0;
}

void display(int y, int *yptr)
{
	char var[7] = "ABCDEF";
	printf("In display() function:\n");
	printf("y value is %d and is stored at address %p.\n", y, &y);
	printf("yptr pointer points to address %p which holds a value of %d.\n", yptr, *yptr);
}
```

Program 1-1

```cpp
// This program calculates the user's pay.
#include <iostream>
using namespace std;

int main()
{
   double hours, rate, pay;

   // Get the number of hours worked.
   cout << "How many hours did you work? ";
   cin >> hours;

   // Get the hourly pay rate.
   cout << "How much do you get paid per hour? ";
   cin >> rate;

   // Calculate the pay.
   pay = hours * rate;

   // Display the pay.
   cout << "You have earned $" << pay << endl;
   return 0;
}
```

Program Output with Example Input Shown in Bold

```
How many hours did you work? 10 [Enter]
How much do you get paid per hour? 15 [Enter]
You have earned $150
```



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

int main()
{
   cout<<"Hello World\n";
   return 0;
}
```

Table 2-1 Special Characters

Character	Name	Description
`//`	Double slash	Marks the beginning of a comment.
`#`	Pound sign	Marks the beginning of a preprocessor directive.
`< >`	Opening and closing brackets	Encloses a filename when used with the `#include` directive.
`( )`	Opening and closing parentheses	Used in naming a function, as in `int main()`
`{ }`	Opening and closing braces	Encloses a group of statements, such as the contents of a function.
`" "`	Opening and closing quotation marks	Encloses a string of characters, such as a message that is to be printed on the screen.
`;`	Semicolon	Marks the end of a complete programming statement.

C++ keywords

This is a list of reserved keywords in C++. Since they are used by the language, these keywords are not available for re-definition or overloading.

`alignas` (since C++11)	`enum`	`return`
`alignof` (since C++11)	`explicit`	`short`
`and`	`export`(1)	`signed`
`and_eq`	`extern`	`sizeof`
`asm`	`false`	`static`
`auto`(1)	`float`	`static_assert` (since C++11)
`bitand`	`for`	`static_cast`
`bitor`	`friend`	`struct`
`bool`	`goto`	`switch`
`break`	`if`	`template`
`case`	`inline`	`this`
`catch`	`int`	`thread_local` (since C++11)
`char`	`long`	`throw`
`char16_t` (since C++11)	`mutable`	`true`
`char32_t` (since C++11)	`namespace`	`try`
`class`	`new`	`typedef`
`compl`	`noexcept` (since C++11)	`typeid`
`const`	`not`	`typename`
`constexpr` (since C++11)	`not_eq`	`union`
`const_cast`	`nullptr` (since C++11)	`unsigned`
`continue`	`operator`	`using`(1)
`decltype` (since C++11)	`or`	`virtual`
`default`(1)	`or_eq`	`void`
`delete`(1)	`private`	`volatile`
`do`	`protected`	`wchar_t`
`double`	`public`	`while`
`dynamic_cast`	`register`	`xor`
`else`	`reinterpret_cast`	`xor_eq`

- (1) - meaning changed in C++11

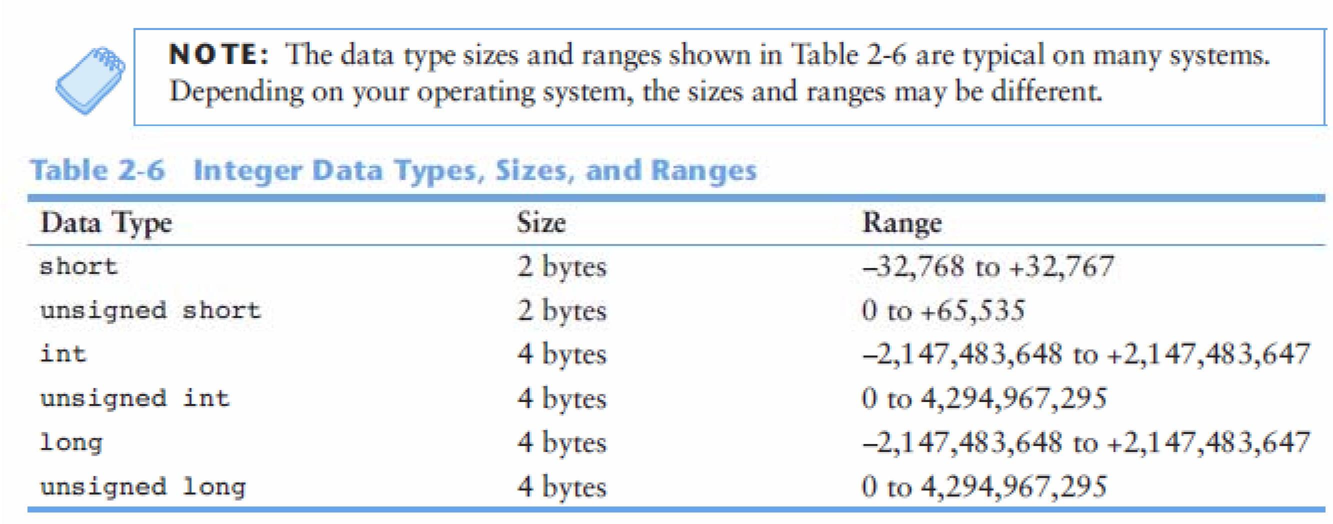
In addition to keywords, there are two *identifiers with special meaning*, which may be used as names of objects or functions, but have special meaning in certain contexts.

`override` (C++11)
`final` (C++11)

Operators in C++ and operator precedence

Level	Operator	Description	Grouping
1	::	scope	Left-to-right
2	() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid	postfix	Left-to-right
3	++ -- ~ ! sizeof new delete	unary (prefix)	Right-to-left
	* &	indirection and reference (pointers)	
	+ -	unary sign operator	
4	(type)	type casting	Right-to-left
5	.* ->*	pointer-to-member	Left-to-right
6	* / %	multiplicative	Left-to-right
7	+ -	additive	Left-to-right
8	<< >>	shift	Left-to-right
9	< > <= >=	relational	Left-to-right
10	== !=	equality	Left-to-right
11	&	bitwise AND	Left-to-right
12	^	bitwise XOR	Left-to-right
13	|	bitwise OR	Left-to-right
14	&&	logical AND	Left-to-right
15	||	logical OR	Left-to-right
16	?:	conditional	Right-to-left
17	= *= /= %= += -= >>= <<= &= ^= |=	assignment	Right-to-left
18	,	comma	Left-to-right

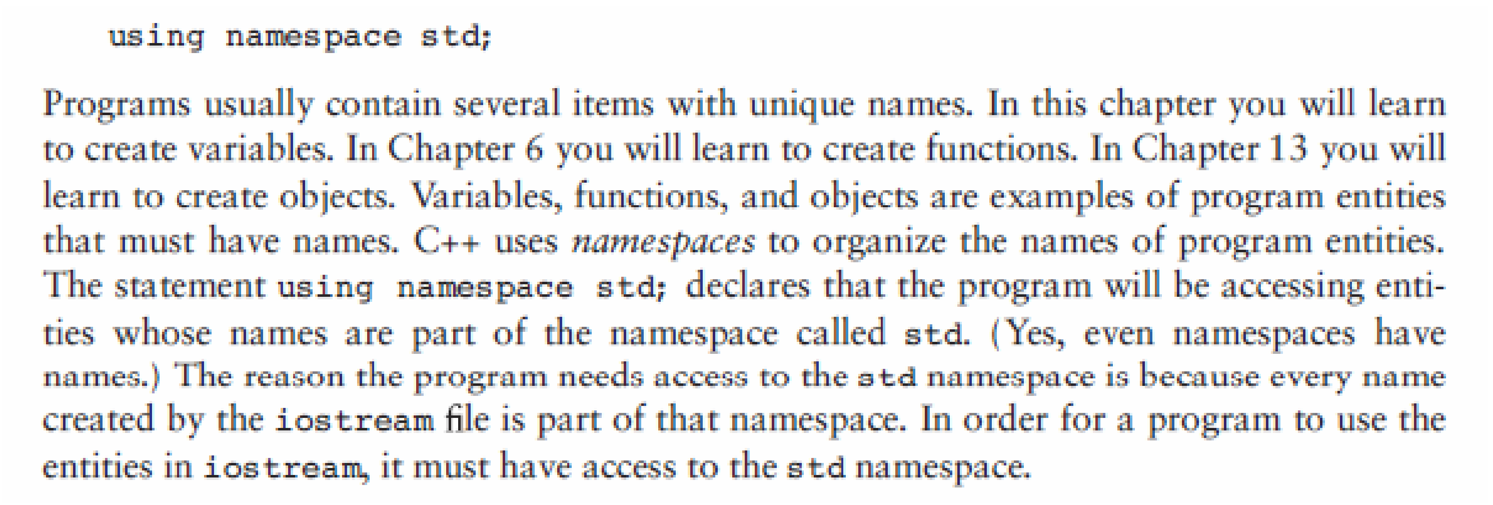
More online at: Operator Precedence and Associativity
msdn.microsoft.com/en-us/library/126fe14k.aspx



NOTE: The data type sizes and ranges shown in Table 2-6 are typical on many systems. Depending on your operating system, the sizes and ranges may be different.

Table 2-6 Integer Data Types, Sizes, and Ranges

Data Type	Size	Range
`short`	2 bytes	–32,768 to +32,767
`unsigned short`	2 bytes	0 to +65,535
`int`	4 bytes	–2,147,483,648 to +2,147,483,647
`unsigned int`	4 bytes	0 to 4,294,967,295
`long`	4 bytes	–2,147,483,648 to +2,147,483,647
`unsigned long`	4 bytes	0 to 4,294,967,295



```
    using namespace std;
```

Programs usually contain several items with unique names. In this chapter you will learn to create variables. In Chapter 6 you will learn to create functions. In Chapter 13 you will learn to create objects. Variables, functions, and objects are examples of program entities that must have names. C++ uses *namespaces* to organize the names of program entities. The statement `using namespace std;` declares that the program will be accessing entities whose names are part of the namespace called `std`. (Yes, even namespaces have names.) The reason the program needs access to the `std` namespace is because every name created by the `iostream` file is part of that namespace. In order for a program to use the entities in `iostream`, it must have access to the `std` namespace.

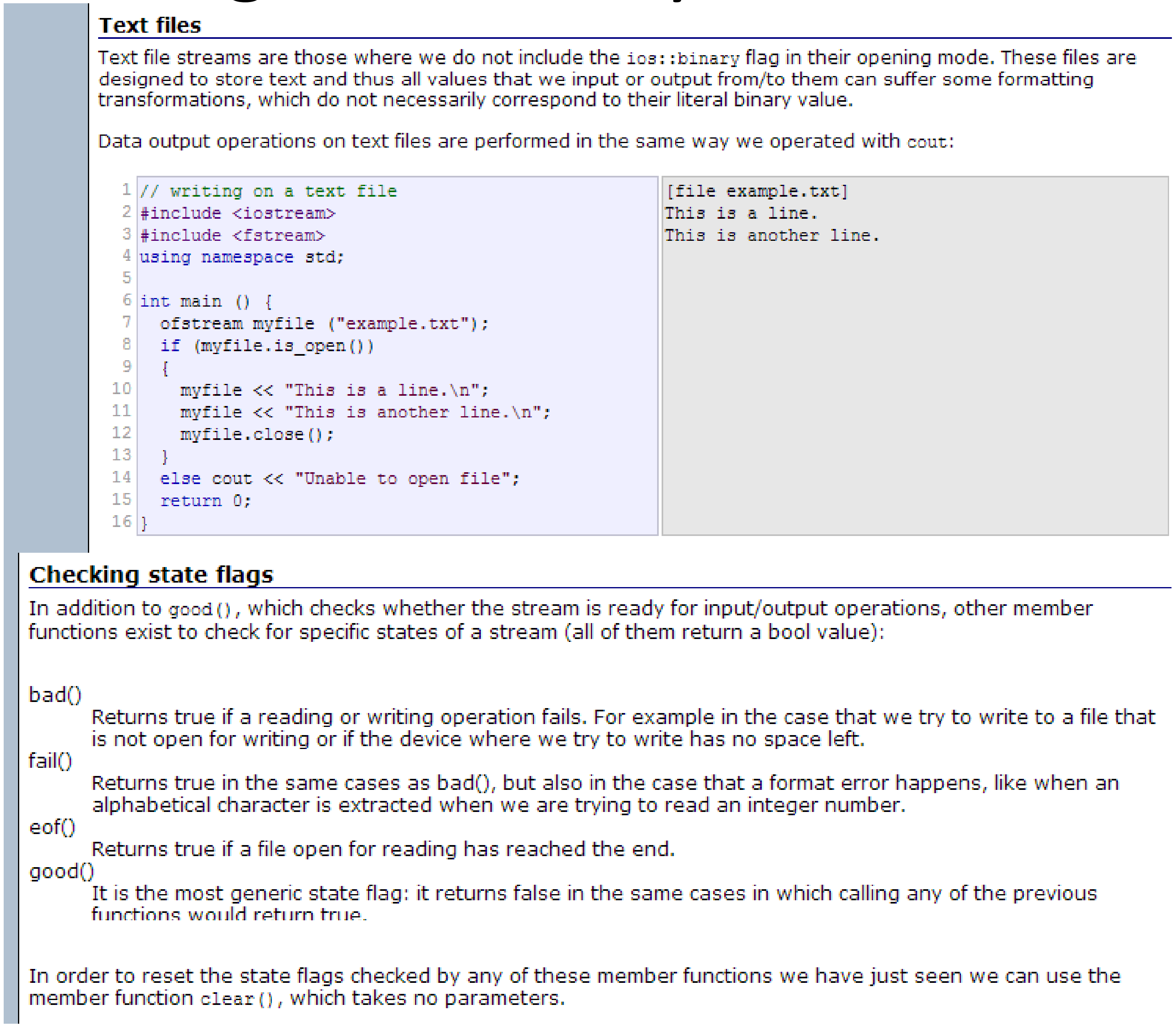
Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name.

```
#include "stdafx.h"
#include <iostream>
#include <string>
using namespace std;
int exit();
class Car
{
	private:
		int Year;
		string Make;
		int Speed;
	public:
		Car(int, string, int);
		int getYear();
		string getMake();
		int getSpeed();
		int Accelerate();
		void getDisplay();
		int Brake();
};
Car::Car(int Yr = 0, string Mk = " ", int Spd = 0) {
Year = Yr;
Make = Mk;
Speed = Spd;
}
int Car::getYear()
{
	cout << "Enter the year of the car: ";
	cin >> Year;
	return Year;
}
string Car::getMake()
{
	cout << "Enter the make and model of the car: 
";
	cin >> Make;
	return Make;
}
int Car::getSpeed()
{
	cout << "Enter the speed of the car: ";
	cin >> Speed;
	return Speed;
}
```

```
int Car::Accelerate()
{
	Speed = Speed + 5;
	return Speed;
}
int Car::Brake()
{
	Speed = Speed - 5;
	return Speed;
}
void Car::getDisplay()
{
	int choice;

	cout << "The car is a " << getYear() << " " << 
getMake() << " going " << getSpeed() << " MPH." << endl;
	Car car(getYear(), getMake(), getSpeed());
	do
	{
		cout << "          Menu         " << endl;
		cout << "-----------------------" << endl;
		cout << " 1. Accelerate         " << endl;
		cout << " 2. Brake              " << endl;
		cout << " 3. Exit               " << endl;
		cout << "-----------------------" << endl;
		cout << "\nEnter your choice:   " << endl;
		cin >> choice;
```

```cpp
                switch (choice)
                {
                        case 1: cout << "Accelerating";
                                        cout << car.Accelerate();
                                        break;
                        case 2: cout << "Braking";
                                        cout << car.Brake();
                                        break;
                        case 3: cout << "Exiting Program";
                                        exit();
                                        break;

                while (choice < 1 || choice > 3)
                {
                        cout << "\nYour choice must be 1-3. Re-enter.\n" << endl;
                        cout << "          Menu          " << endl;
                        cout << "------------------------" << endl;
                        cout << " 1. Accelerate          " << endl;
                        cout << " 2. Brake               " << endl;
                        cout << " 3. Exit                " << endl;
                        cout << "------------------------" << endl;
                        cout << "\nEnter your choice:   " << endl;
                        cin >> choice;
                }
        } while (choice != 3);
}
int exit()
{
        return (0);
}
int main()
{
        getDisplay();
        cout << endl << "Press ENTER to exit...";
        cin.clear();
        cin.sync();
        cin.get();
        return 0;
}
```

Reading from binary and text files



Text files

Text file streams are those where we do not include the `ios::binary` flag in their opening mode. These files are designed to store text and thus all values that we input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

Data output operations on text files are performed in the same way we operated with `cout`:

```
// writing on a text file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile ("example.txt");
  if (myfile.is_open())
  {
    myfile << "This is a line.\n";
    myfile << "This is another line.\n";
    myfile.close();
  }
  else cout << "Unable to open file";
  return 0;
}
```

```
[file example.txt]
This is a line.
This is another line.
```

Checking state flags

In addition to `good()`, which checks whether the stream is ready for input/output operations, other member functions exist to check for specific states of a stream (all of them return a bool value):

bad()
: Returns true if a reading or writing operation fails. For example in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.

fail()
: Returns true in the same cases as bad(), but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.

eof()
: Returns true if a file open for reading has reached the end.

good()
: It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true.

In order to reset the state flags checked by any of these member functions we have just seen we can use the member function `clear()`, which takes no parameters.

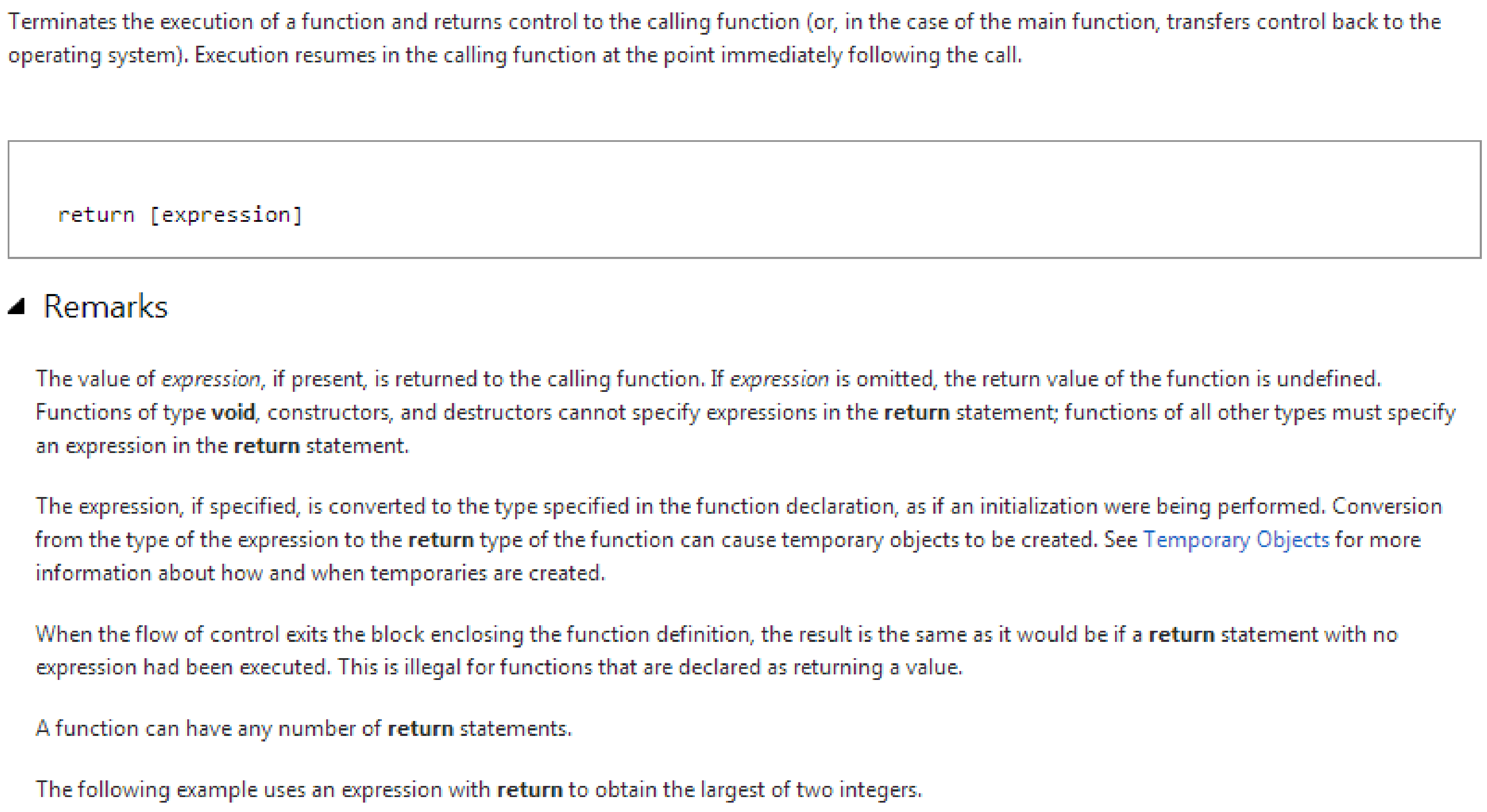
More at: http://www.cplusplus.com/doc/tutorial/files/

Data input from a file can also be performed in the same way that we did with `cin`:

```cpp
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  string line;
  ifstream myfile ("example.txt");
  if (myfile.is_open())
  {
    while ( myfile.good() )
    {
      getline (myfile,line);
      cout << line << endl;
    }
    myfile.close();
  }

  else cout << "Unable to open file";

  return 0;
}
```

```
This is a line.
This is another line.
```

Return statement and functions



Terminates the execution of a function and returns control to the calling function (or, in the case of the main function, transfers control back to the operating system). Execution resumes in the calling function at the point immediately following the call.

```
return [expression]
```

Remarks

The value of *expression*, if present, is returned to the calling function. If *expression* is omitted, the return value of the function is undefined. Functions of type **void**, constructors, and destructors cannot specify expressions in the **return** statement; functions of all other types must specify an expression in the **return** statement.

The expression, if specified, is converted to the type specified in the function declaration, as if an initialization were being performed. Conversion from the type of the expression to the **return** type of the function can cause temporary objects to be created. See Temporary Objects for more information about how and when temporaries are created.

When the flow of control exits the block enclosing the function definition, the result is the same as it would be if a **return** statement with no expression had been executed. This is illegal for functions that are declared as returning a value.

A function can have any number of **return** statements.

The following example uses an expression with **return** to obtain the largest of two integers.

◢ Example

```
// return_statement2.cpp
#include <stdio.h>

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

int main()
{
    int nOne = 5;
    int nTwo = 7;

    printf_s("\n%d is bigger\n", max( nOne, nTwo ));
}
```

Control Structures

- Conditional structures
- Iteration structures
- Jump statements
- Selective structure

Conditional structure: if and else

The `if` keyword is used to execute a statement or block only if a condition is fulfilled. Its form is:

```
if (condition) statement
```

Where `condition` is the expression that is being evaluated. If this condition is true, `statement` is executed. If it is false, `statement` is ignored (not executed) and the program continues right after this conditional structure.
For example, the following code fragment prints `x is 100` only if the value stored in the `x` variable is indeed `100`:

```
if (x == 100)
  cout << "x is 100";
```

If we want more than a single statement to be executed in case that the condition is true we can specify a block using braces `{ }`:

```
if (x == 100)
{
   cout << "x is ";
   cout << x;
}
```

We can additionally specify what we want to happen if the condition is not fulfilled by using the keyword `else`. Its form used in conjunction with `if` is:

```
if (condition) statement1 else statement2
```

For example:

```
if (x == 100)
  cout << "x is 100";
else
  cout << "x is not 100";
```

prints on the screen `x is 100` if indeed `x` has a value of `100`, but if it has not -and only if not- it prints out `x is not 100`.

The `if + else` structures can be concatenated with the intention of verifying a range of values. The following example shows its use telling if the value currently stored in `x` is positive, negative or none of them (i.e. zero):

```
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
```

Remember that in case that we want more than a single statement to be executed, we must group them in a block by enclosing them in braces `{ }`.

Iteration structures (loops)

Loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled.

The while loop

Its format is:

```
while (expression) statement
```

and its functionality is simply to repeat statement while the condition set in expression is true.
For example, we are going to make a program to countdown using a while-loop:

```
// custom countdown using while

#include <iostream>
using namespace std;

int main ()
{
  int n;
  cout << "Enter the starting number > ";
  cin >> n;

  while (n>0) {
    cout << n << ", ";
    --n;
  }

  cout << "FIRE!\n";
  return 0;
}
```

```
Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

When the program starts the user is prompted to insert a starting number for the countdown. Then the `while` loop begins, if the value entered by the user fulfills the condition `n>0` (that `n` is greater than zero) the block that follows the condition will be executed and repeated while the condition (`n>0`) remains being true.

The do-while loop

Its format is:

```
do statement while (condition);
```

Its functionality is exactly the same as the while loop, except that `condition` in the do-while loop is evaluated after the execution of statement instead of before, granting at least one execution of `statement` even if `condition` is never fulfilled. For example, the following example program echoes any number you enter until you enter `0`.

```cpp
// number echoer

#include <iostream>
using namespace std;

int main ()
{
  unsigned long n;
  do {
    cout << "Enter number (0 to end): ";
    cin >> n;
    cout << "You entered: " << n << "\n";
  } while (n != 0);
  return 0;
}
```

```
Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0
```

The do-while loop is usually used when the condition that has to determine the end of the loop is determined **within** the loop statement itself, like in the previous case, where the user input within the block is what is used to determine if the loop has to end. In fact if you never enter the value `0` in the previous example you can be prompted for more numbers forever.

The for loop

Its format is:

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

and its main function is to repeat `statement` while `condition` remains true, like the while loop. But in addition, the `for` loop provides specific locations to contain an `initialization` statement and an `increase` statement. So this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration.

It works in the following way:

1. `initialization` is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. `condition` is checked. If it is true the loop continues, otherwise the loop ends and `statement` is skipped (not executed).
3. `statement` is executed. As usual, it can be either a single statement or a block enclosed in braces `{ }`.
4. finally, whatever is specified in the `increase` field is executed and the loop gets back to step 2.

Here is an example of countdown using a for loop:

```
// countdown using a for loop
#include <iostream>
using namespace std;
int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "FIRE!\n";
  return 0;
}
```

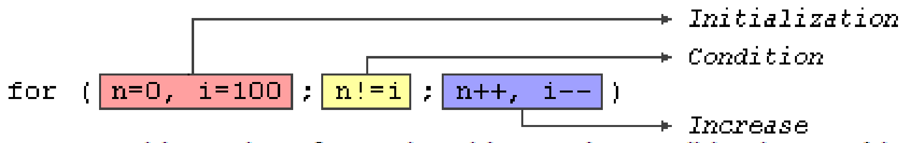
```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

The `initialization` and `increase` fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write: `for (;n<10;)` if we wanted to specify no initialization and no increase; or `for (;n<10;n++)` if we wanted to include an increase field but no initialization (maybe because the variable was already initialized before).

Optionally, using the comma operator (`,`) we can specify more than one expression in any of the fields included in a `for` loop, like in `initialization`, for example. The comma operator (`,`) is an expression separator, it serves to separate more than one expression where only one is generally expected. For example, suppose that we wanted to initialize more than one variable in our loop:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
   // whatever here...
}
```

This loop will execute for 50 times if neither `n` or `i` are modified within the loop:



```
for ( n=0, i=100 ; n!=i ; n++, i-- )
```

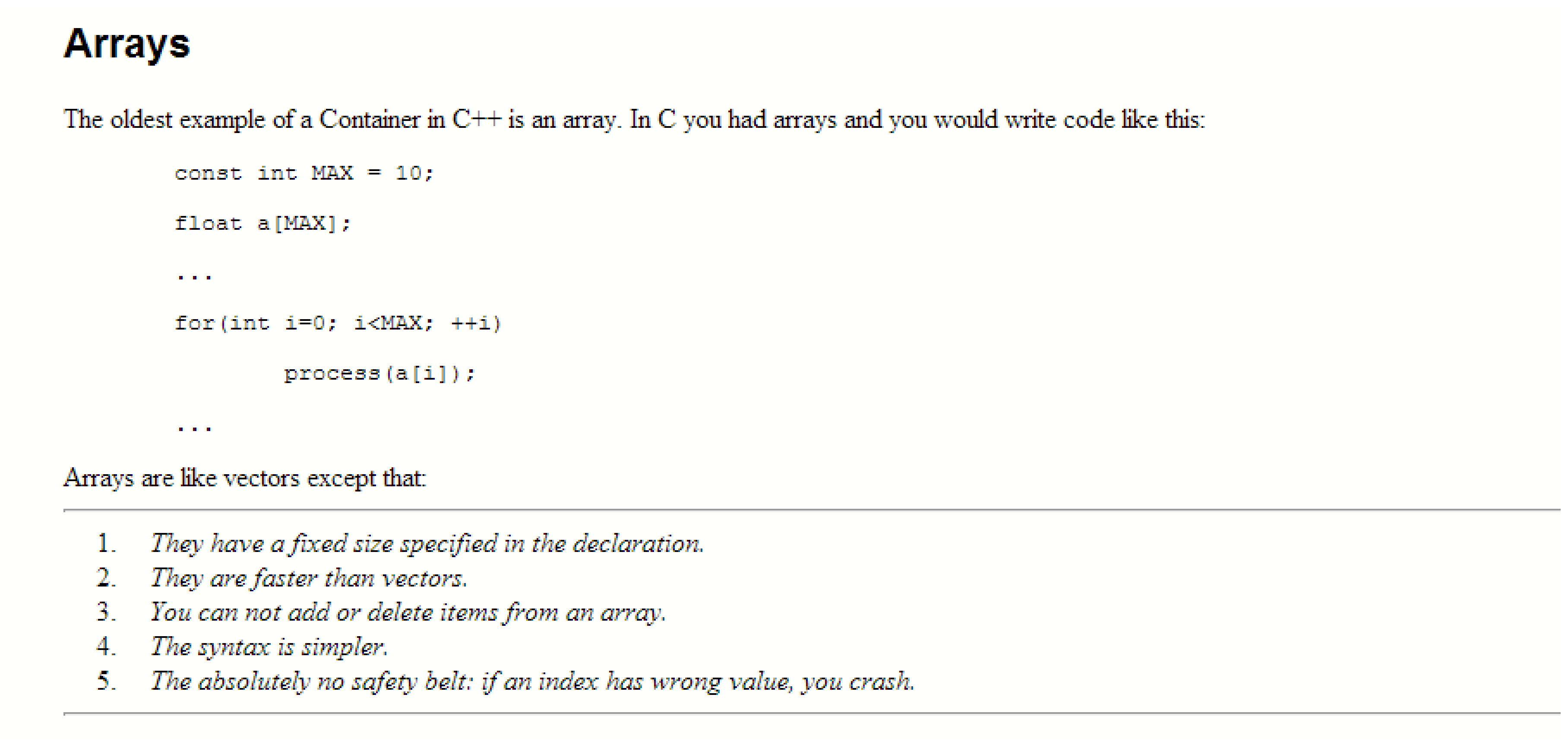
`n` starts with a value of `0`, and `i` with `100`, the condition is `n!=i` (that `n` is not equal to `i`). Because `n` is increased by one and `i` decreased by one, the loop's condition will become false after the 50th loop, when both `n` and `i` will be equal to `50`.

Jump statements.

The break statement

Using `break` we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before its natural end (maybe because of an engine check failure?):

Arrays, Vectors, and Lists



Arrays

The oldest example of a Container in C++ is an array. In C you had arrays and you would write code like this:

```
const int MAX = 10;

float a[MAX];

...

for(int i=0; i<MAX; ++i)

        process(a[i]);

...
```

Arrays are like vectors except that:

1. *They have a fixed size specified in the declaration.*
2. *They are faster than vectors.*
3. *You can not add or delete items from an array.*
4. *The syntax is simpler.*
5. *The absolutely no safety belt: if an index has wrong value, you crash.*

C++ Array Size: The Size of an Array in C++
Short answer: use **sizeof(array)/sizeof(element type)**.
For example: sizeof(scores)/sizeof(int), where scores is an array of ints.
This does not work with pointers.

```cpp
#include <string>
#include <algorithm>
#include <iostream>
using namespace std;
bool sort_strings(const string& a, const string& b)
{
             return a < b;
}
 int main(int argc, char *argv[])
 {
             // Declare and initialize an array of strings.
             string strings[] = {
                         "Fox",
                         "Beetroot",
                         "Zebra",
                         "Aardvark",
                         "Cabbage"
             };
             // Sort the array with STL sort.
             sort(strings, strings + 5, sort_strings);
             for(int i=0; i < sizeof(strings)/sizeof(string); i++)
             {
                         cout << strings[i] << endl;
             }
 }
```

```
OUTPUT:
Aardvark
Beetroot
Cabbage
Fox
Zebra
```

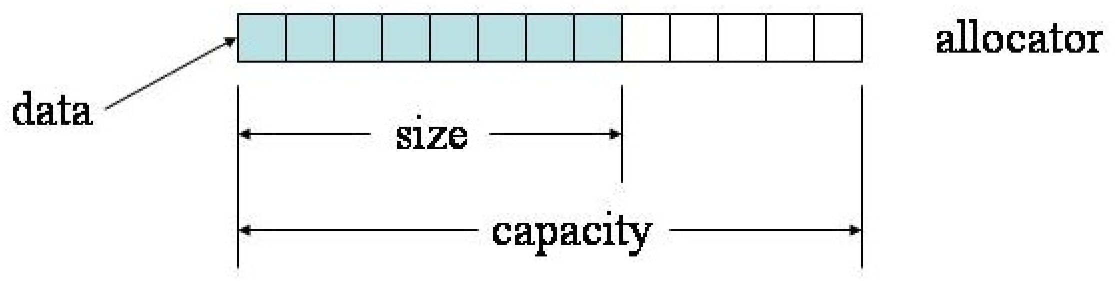
```cpp
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
// Multidimensional array of numbers.
// We need to specify sizes for
// all except the first dimension.
int numbers[][3] =
{
            {0, 1, 2},
            {3, 4, 5}
};
for(int row=0; row < 2; row++){
    for(int col=0; col < 3; col++){
       cout << numbers[row][col] << endl;
    }
}
}
```

```
OUTPUT:
0
1
2
3
4
5
```

Arrays, Vectors, and Lists

Vectors

The STL vector class provides functionality similar to that of C arrays. Vectors are contiguous array of elements where the the first "size" elements are constructed (initialized) and the last "capacity - size" elements are uninitialized.



Vectors may be implemented by providing a pointer to an array allocated on the heap, but this is not guaranteed by the standard. Unlike C arrays, vectors are expandable. Expanding a vector beyond its current capacity causes the vector elements to be copied to a larger array also allocated on the heap.

Vectors share a major vulnerability with arrays; as vectors perform no bounds-checking on `operator[]`. However, the `vector::at()` method performs index retrieval and assignment with range checking; it throws an `out_of_range` exception if the index provided is outside the vector's range.

A vector may be initialized directly by the constructor:

```
vector<int> dat( ARRAY_SIZE, 42); // set all elements to 42
```

or by the `fill` algorithm:

```
fill( dat.begin(), dat.end(), 42); // set all elements to 42
```

A vector may be initialized directly by the constructor:

```
vector<int> dat( ARRAY_SIZE, 42); // set all elements to 42
```

or by the `fill` algorithm:

```
fill( dat.begin(), dat.end(), 42); // set all elements to 42
```

or directly by indices:

```
for (size_t i = 0; i < dat.size(); i++) {
  dat[i] = 42; /* Assigns 42 to each element; */
  /* ... */
}
```

Hint: double-click to select code

or by using iterators:

```
for (vector<int>::iterator i = dat.begin(); i != dat.end(); i++) {
   *i = 42; /* Assigns 42 to each element; */
}
```

The relationship between array subscripts and pointers also applies to vector 'subscripts' and iterators.

```
/*			Suppose that we want to input an unknown number of numbers and then
			print them out forwards and then backwards, using a vector. We will
			push ints onto the back of a vector called v. We will then print
			each item in v in turn. Finally we will print the vector backwards.
			You can download the code from [ Vector.cpp ] but here are the
			highlights. First we must declare the facilities we want to use */
#include <iostream>
#include <vector>
using namespace std;
void print( const vector<int>& ) ;//utility function outputs a vector of ints
void print_backwards( const vector<int> &);
/*Then we describe the main program: */
int main() {
			vector<int> v;
			int number;
			cout <<"Input some numbers and then end the input\n";
			while(cin>>number){
						v.push_back(number);
			}//while(more)
			print(v);
			print_backwards(v);
}//main
/* Finally the two procedures that print out the data: */
void print_backwards( const vector<int> &a) {
			for(int i=a.size()-1; i>=0; --i)
						cout << a[i] << " ";
			cout << endl;
			cout << "----------------"<<endl;
}//print_backwards
void print( const vector<int>& a)
{
			for(int i=0; i<a.size(); ++i)
						cout << a[i] << " ";
			cout << endl;
			cout << "----------------"<<endl;
}//print
```

Vectors are good when we have an unknown sequence of similar items to store and we want to access them by their sequence numbers.

Vectors are held in a special library and can be used in a file that has

```
                    #include <vector>
```

at its beginning. It uses the "std" namespace so you will need "using ...;" statements.
Table

Declaration	vector<type> v(initial_size);
Accessors	v.empty(), v.size(), v.front(), v.back()
Mutators	v.push_back(T), v.pop_back()
Operators	v[int], v.at(int), v1=v2;, v1==v2

(Close Table)

Key Facts

You need to remember the following facts about vectors:

1. *A vector is an object that contains a sequence of other objects inside it.*
2. *The objects inside must all take up the same amount of storage.*
3. *They are numbered starting with 0.*
4. If the whole vector is called *v* then the items in it are written *v[0]*, *v[1]*, *v[2]*, ...
5. The last item is *v[v.size()-1]* NOT *v[v.size()]*.
6. New items can be "pushed" onto the end of the vector.
7. The last item can be "popped" off of a vector.
8. Vectors can therefore change size.
9. We can find out the current size of a vector: v.size()
10. Vectors can be empty. If so v.empty() is true.
11. If a vector is empty then v[i] and v.pop.... crash.
12. Vectors are empty. by default, when created.
13. Vectors shouls be passed by reference whenever possible.

Details

Suppose that *T* is any type or class - say int, float, double, or the name of a class, then

```
vector<T> v;
```

declares a new and empty **vector** called *v*. Given object *v* declare like the above you can do the following things with it:

- *test to see if v is empty:*

  ```
  v.empty()
  ```

- *find how many items are in v:*

  ```
  v.size()
  ```

- *push t in T onto the end of v:*

  ```
  v.push_back(t)
  ```

- *pop the back of v off v:*

  ```
  v.pop_back()
  ```

- *Access the i'th item (0<=i<size()) without checking to see if it exists:*

  ```
  v[i]
  ```

- *Assign a copy of v1 to v:*

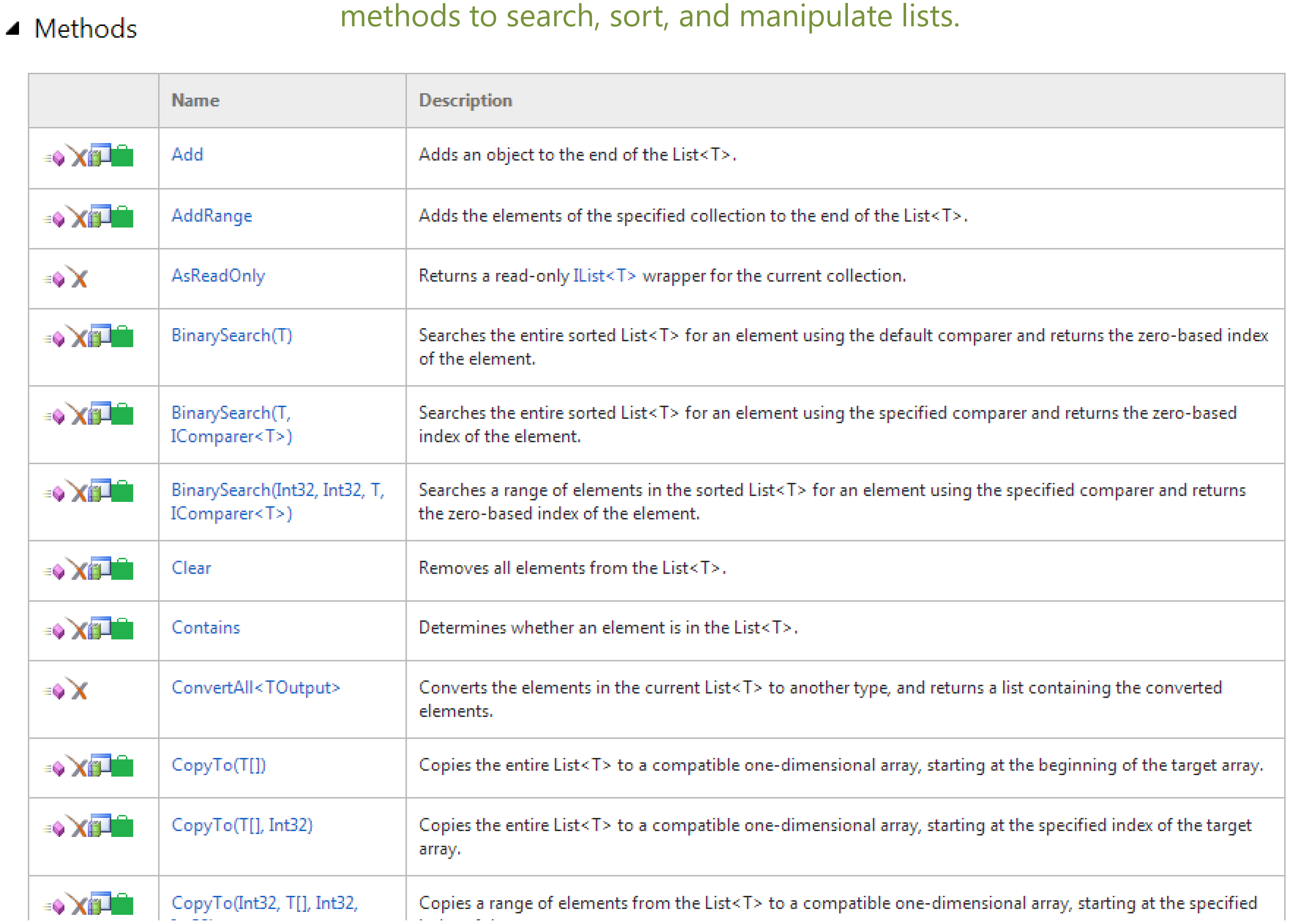
  ```
  v = v1
  ```

List

```
// constructing lists
#include <iostream>
#include <list>

int main ()
{
  // constructors used in the same order as described above:
  std::list<int> first;                                // empty list of ints
  std::list<int> second (4,100);                       // four ints with value 100
  std::list<int> third (second.begin(),second.end());  // iterating through second
  std::list<int> fourth (third);                       // a copy of third

  // the iterator constructor can also be used to construct from arrays:
  int myints[] = {16,2,77,29};
  std::list<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

  std::cout << "The contents of fifth are: ";
  for (std::list<int>::iterator it = fifth.begin(); it != fifth.end(); it++)
    std::cout << *it << ' ';

  std::cout << '\n';

  return 0;
}
```

Using the MSDN to find the methods of class list: List<T> Class (http://msdn.microsoft.com/en-us/library/6sh2ey19.aspx)

Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.

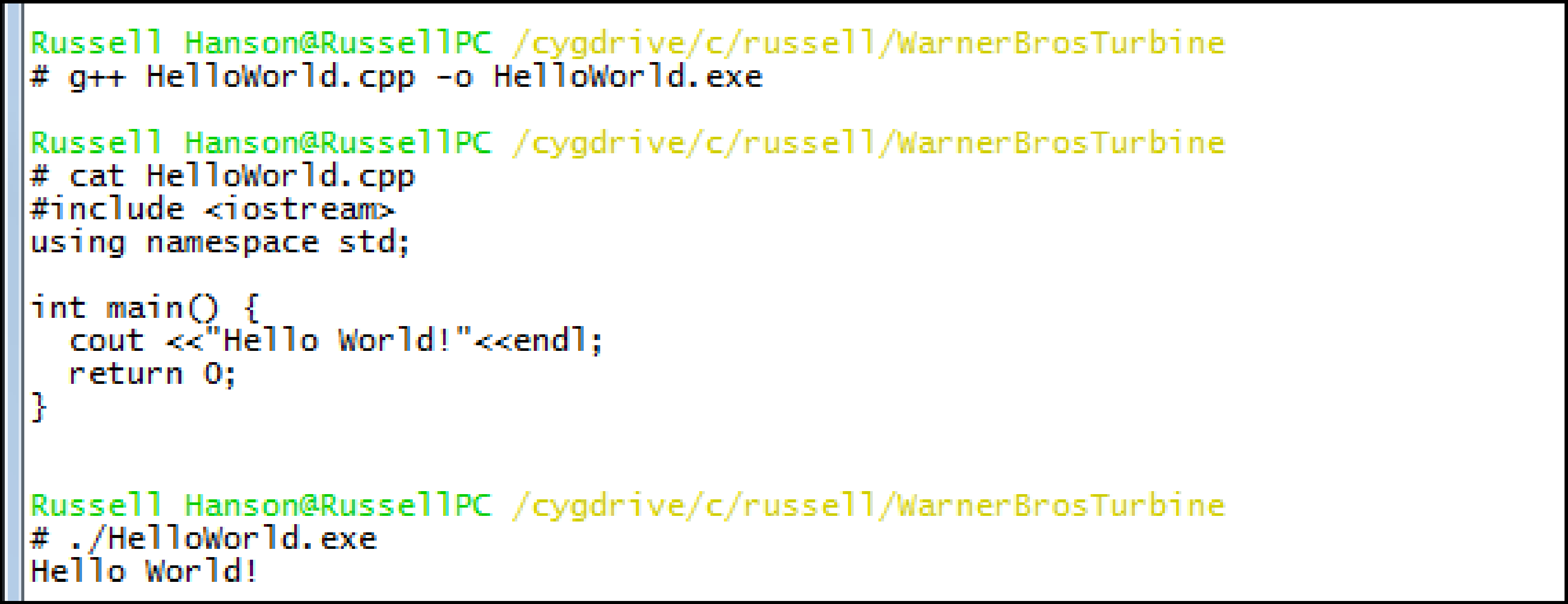


Methods

	Name	Description
	Add	Adds an object to the end of the List<T>.
	AddRange	Adds the elements of the specified collection to the end of the List<T>.
	AsReadOnly	Returns a read-only IList<T> wrapper for the current collection.
	BinarySearch(T)	Searches the entire sorted List<T> for an element using the default comparer and returns the zero-based index of the element.
	BinarySearch(T, IComparer<T>)	Searches the entire sorted List<T> for an element using the specified comparer and returns the zero-based index of the element.
	BinarySearch(Int32, Int32, T, IComparer<T>)	Searches a range of elements in the sorted List<T> for an element using the specified comparer and returns the zero-based index of the element.
	Clear	Removes all elements from the List<T>.
	Contains	Determines whether an element is in the List<T>.
	ConvertAll<TOutput>	Converts the elements in the current List<T> to another type, and returns a list containing the converted elements.
	CopyTo(T[])	Copies the entire List<T> to a compatible one-dimensional array, starting at the beginning of the target array.
	CopyTo(T[], Int32)	Copies the entire List<T> to a compatible one-dimensional array, starting at the specified index of the target array.
	CopyTo(Int32, T[], Int32,	Copies a range of elements from the List<T> to a compatible one-dimensional array, starting at the specified

First Program, “Hello World!”

```cpp
#include <iostream>
using namespace std;
int main() {
cout <<"Hello World!"<<endl;
return 0;
}
```



```
Russell Hanson@RussellPC /cygdrive/c/russell/WarnerBrosTurbine
# g++ HelloWorld.cpp -o HelloWorld.exe

Russell Hanson@RussellPC /cygdrive/c/russell/WarnerBrosTurbine
# cat HelloWorld.cpp
#include <iostream>
using namespace std;

int main() {
  cout <<"Hello World!"<<endl;
  return 0;
}


Russell Hanson@RussellPC /cygdrive/c/russell/WarnerBrosTurbine
# ./HelloWorld.exe
Hello World!
```