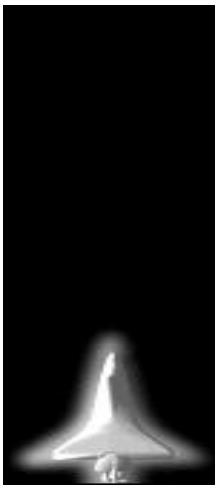
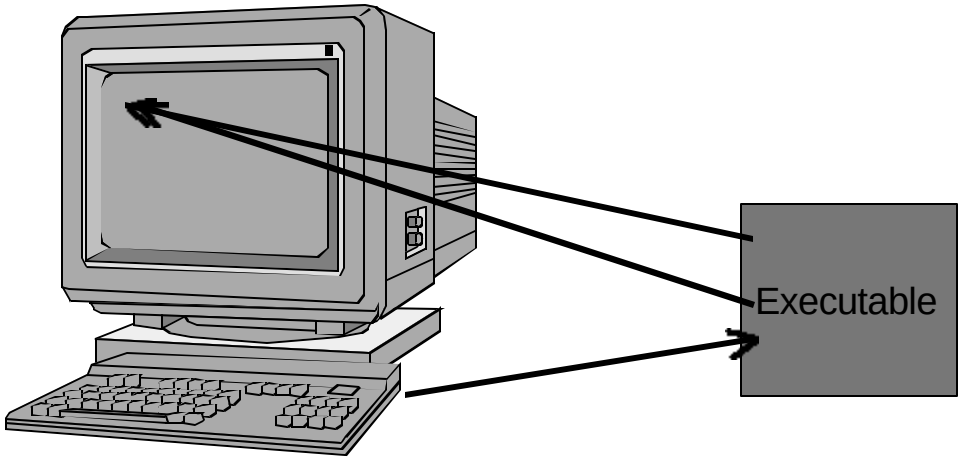
Getting Started



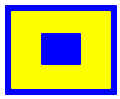
1 Animated GIF from http://www.earthcon.com/clipart/page10.htm.

Overview

- The `main` function
- Include files
- The standard namespace
- Indentation

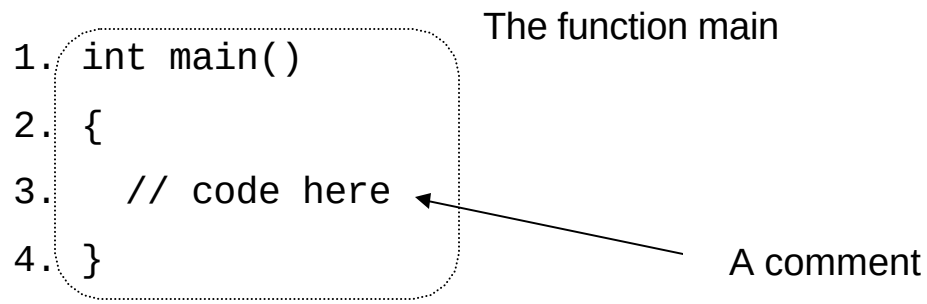


1. In this chapter we will look briefly at the various pieces of the language and standard library you need to know to write a first program. We will come back to most of these to look at them in more detail later in the course.



Main

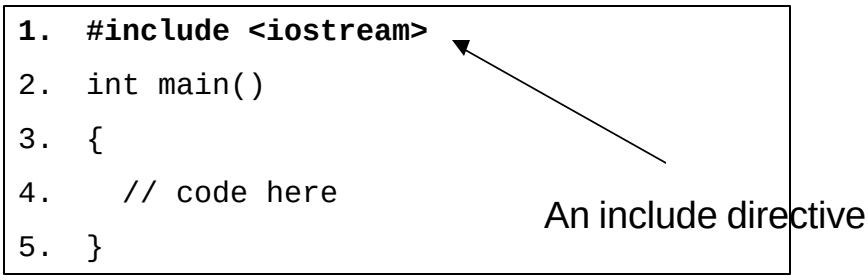
- In C++, every program has a function called `main`
- When the program is run, the function `main` is executed
- Most programs have a `main` function that looks like this:



1. All the code examples in this course are numbered, to make it easier for the instructor or the students to refer to specific lines. You do **not** number the lines in your code.
2. The first line says that `main` is a function that takes no parameters and returns an integer. The return value is used by the operating system to decide whether the program succeeded or failed.
3. `main` is special in that you do not have to explicitly return a value; it defaults to 0, which means success.
4. Functions in C++ have a head and a body.
5. The head says what the return type is, what the name of the function is, and, in parentheses, what parameters the function takes. We will look at function parameters in XXX.
6. The body of a function is enclosed in curly braces { and }, and consists of a sequence of instructions that are executed in the order they are given.
7. Comments are information for people reading or maintaining (correcting or improving) the code; they are ignored by the compiler.
8. There are two kinds of comments in C++:
   - C-style comments, starting with a /*, and ending with */
   - C++ comments, starting with a // and ending at the end of the line.
9. C-style comments are useful for multiple-line comments; C++ comments are useful for short comments.
10. There is another form of main that allows you to pass parameters to main; this is useful for console-based programs (as opposed to programs that use a graphical user interface); we will see this in XXX.
11. In some development environments (e.g., Windows) you will not write the main function yourself; it is built by the development environment. Instead, you define a function called WinMain, which will be called after various initializations have been done.

Include Files

- C++ consists of a *language* and a *standard library*
- To use a library, you include a *header file* that contains *declarations* that tell the compiler about the library
- Including header files is done with a *preprocessor directive*, `#include`, *e.G.*,



1. A library is a bundle of software provided by your compiler vendor or a third-party company.
2. C++ compilers are (since 1998 or so) always provided with the standard C++ library. Using the standard library makes code more portable (to different platforms) and easier to maintain, as most C++ programmers know it.
3. The preprocessor is part of the compiler; it is a program that takes a file and replaces #include directives with the contents of the file specified. It can also be used to make macro substitutions, but this is generally unwise, as there are better alternatives in almost every case.
4. Preprocessor directives must start in the left-most column. This, and // comments, are the only cases where there is such a restriction; generally, C++ treats all whitespace equivalently. For example, the program shown here is exactly equivalent to:
   - #include <iostream>
   - Int main(){//code here
   - }
5. An include directive specifies the name of the include file in angle brackets if the file is part of the standard library; if it is not, the name is specified in double quotes, e.g., #include “myfile.h”
6. Standard library includes do not have an extension --- this is because different platforms use different conventions for naming header files.
7. This is a complete C++ program, so it is shown in a solid box.

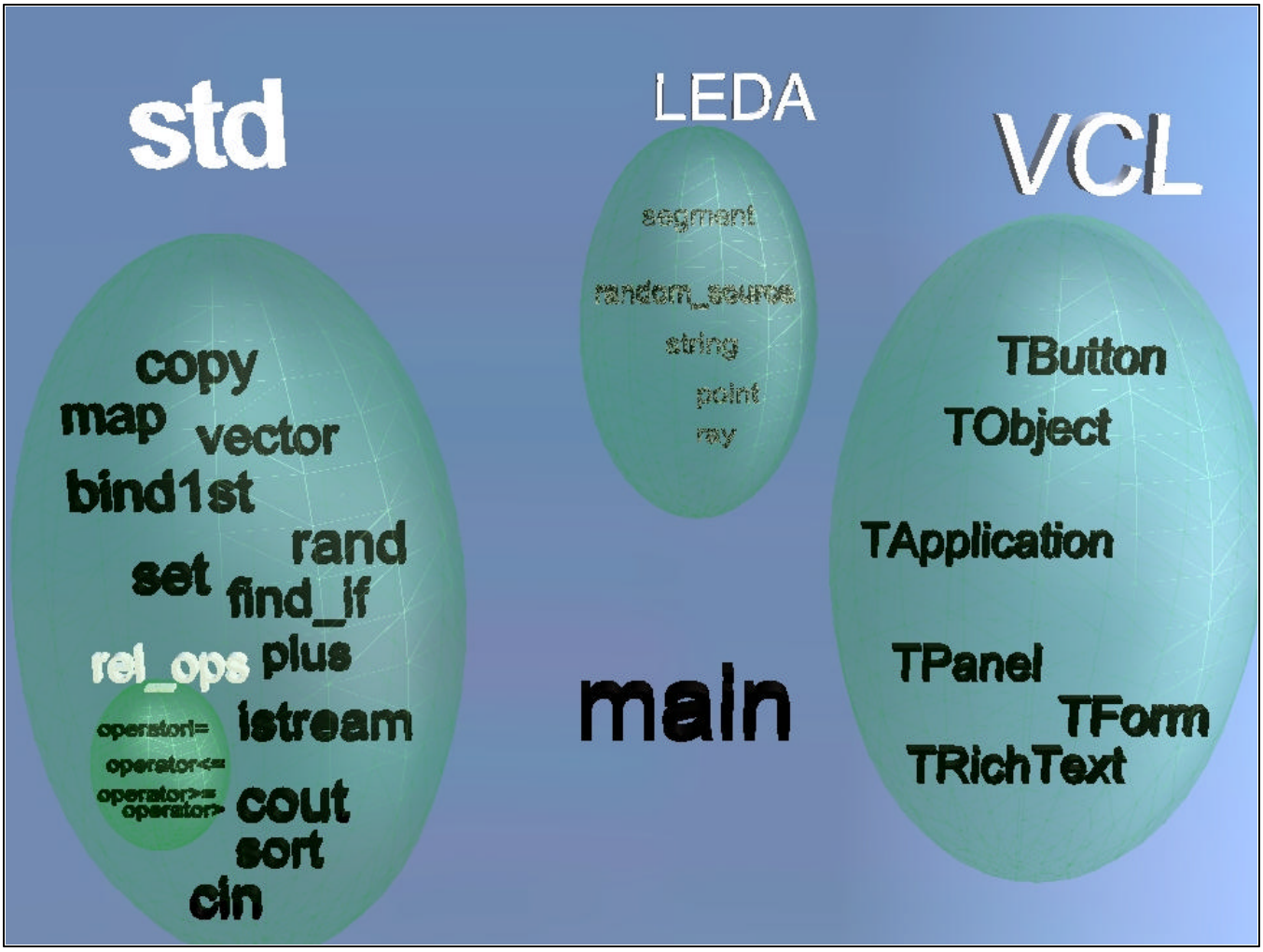
The Standard Namespace

- *Namespaces* provide a way of dividing up a piece of software into pieces without *name collisions*
- Everything in the standard library is in a namespace called `std`
- A *namespace directive* tells the compiler to look for names in that namespace

```
1.  #include <iostream>
2.  int main()
3.  {
4.    using namespace std;
5.    // code here
6.  }
```

using directive

1. Namespaces were added to C++ fairly late in the standardization process, so they are unfamiliar to many C++ programmers who learned C++ before they were available.
2. Some compilers still do not fully support namespaces.
3. A name collision is the problem that occurs when the same name is used for different things in two parts of a program.
4. If the same name is used in two different namespaces, there is no problem.
5. We will look at namespaces in more detail in XXX; for now, all you need to know is to put the line using namespace std at the beginning of each function that uses components of the standard library.



1. A namespace is a piece of code whose names are isolated from other namespaces.
2. You can use names in a namespace by explicitly qualifying them, like std::cout, with the namespace name, '::', referred to as a scoping operator, and the name in that namespace.
3. Here is a picture of three namespaces, std (the ANSI standard C++ library), LEDA (a free(?) library providing classes for mathematical constructs like geometric shapes, and VCL, the Borland visual component library.
4. As of writing (Jan 2000), Leda is not packaged like this, and neither is Borland's VCL. The gnu compilers do not properly support namespaces.
5. The names in the std namespace are precisely defined by the ANSI specification. Those in LEDA and VCL may (and typically do) change from version to version. The Borland convention for naming types is with an initial T. Microsoft uses C. It doesn't matter what you use, just be consistent.
6. One common convention is to name all user-defined types with "cuddly caps": GraphNode, SetInts, etc.
7. Another is that preprocessor macros shout at you: int here; int is; int a = MAX(here,is);
8. (for the curious, the result of the above is undefined – variables aren't iniitialized if you don't specify an initial value.)
9. Many companies have coding standards that specify things like this.
10. There are tools that will automatically check that you follow the rules, similar to lint for C.

Namespaces

- In large projects, the problem of name collisions becomes serious
- A name collision is caused by two variables or classes in different modules having the same name.
- To get around this problem, C++ lets you define *namespaces* --- portions of code in which everything declared is implicitly preceded with the name of the namespace.
- There is one namespace in the standard library: `std`, and some nested namespaces such as `rel_ops`.

1. 1 Namespaces are a relatively recent addition to C++; some older compilers may not implement them properly.

Defining a namespace

- The keyword namespace introduces a new namespace; names defined inside it are implicitly qualified with the name of the namespace:

```
1.  namespace XWindows
2.  {
3.    class Window    // really XWindows::Window
4.    {
5.    // …
6.    };
7.    void printWindows();   // really XWindows::printWindows
8.      };
9.  namespace NTWindows
10. {
11.   class Window          // really NTWindows::Window, so no collision
12.   {
13.     // …
14.   };
15. };
```

1. 1 Namespaces can be reopened, unlike classes, so that the components in a namespace can be broken over several files.

Using namespaces

- You can access names in a namespace with the :: qualfier:
- This quickly becomes tedious

```
1.  #include <string>
2.  #include <iostream>

3.  int main()
4.  {
5.    std::string a(10, 'a'); // make string aaaaaaaaaa
6.    std::cout << a << std::endl;
7.  }
```

1. All of the components of the standard library are in the namespace called `std`.
2. The `string` class is defined in the header file `<string>`.
3. We will see strings and IO in more detail shortly.

Importing Namespaces

- The standard library is in the namespace `std`
- There are some nested namespaces in `std`, e.g., `rel_ops`
- You can import an entire namespace:

```
using namespace std;
```

1. 1 Importing the entire `std` namespace may be inadvisable, as it contains a huge number of names.
2. 2 However, for small projects, importing the entire std namespace is unlikely to cause conflicts, and we recommend that you do this for simplicity in the labs.

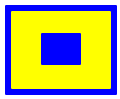
Selective importing

- You can also import individual names:

```
using std::vector;
```

- For small projects, importing the entire `std` namespace is fine, but for larger projects it may be inadvisable

1. 1 There is as yet very little experience in the industry with namespaces, as they have been available only for a few years, so there is not yet a consensus on the best ways to use them.



Indentation

- *Indentation* is used to make code more readable
- It is ignored by the compiler
- Indentation is **extremely** important
- Many *development environments* have facilities that help you indent well
- There are different *indentation conventions*

1. Poorly indented code is very difficult to read.
2. The Emacs editor has excellent indentation facilities –- not only does it automatically indent each line for you, it lets you select a region and automatically indent the entire region. It also lets you specify various parameters to customize the way indentation is done, e.g., how many spaces to indent by, which of several common conventions to use, etc.
3. Most PC development environments (e.g., Visual C++, Borland Builder) have some indentation facilities, but they usually do not allow you to indent a region automatically.
4. There are different conventions for indenting --- it is more important that you be consistent than which convention you use.
5. Many companies have coding standards that specify how you should indent your code.

Good Indentation

- Shows the logical structure

```
1.  int main()
2.  {
3.      while (true)
4.      {
5.          int n;
6.          cin >> n;
7.          if (n < 0)
8.          {
9.              cout << "negative" << endl;
10.         }
11.         else
12.             cout << "positive" << endl;
13.         if (n == 0)
14.             break;
15.     }
16.     cout << "done loop" << endl;
17. }
```

1. The indentation is exagerated here for illustration, usually indenting two spaces is enough.
2. Personally, I prefer to have the beginning and closing braces in the same column, to make it easy to see the blocks. Other people dislike wasting so many lines with only an open or a closed brace on them, so they would write this:

```
int main() {
    while (true)  {
        int n;
        cin >> n;
        if (n < 0)  {
            cout << "negative" << endl;
        }
        else
            cout << "positive" << endl;
        if (n == 0)
            break;
    }
```

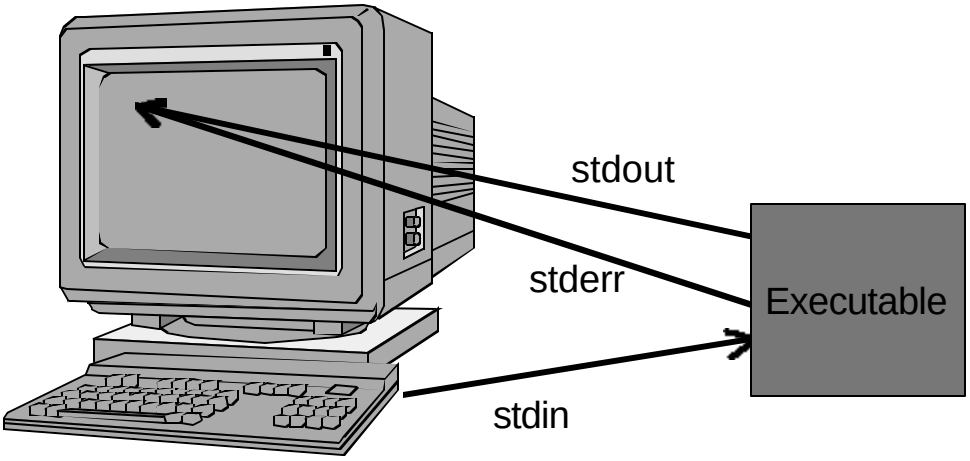
Poor Indentation

```
1.  int main()
2.  {
3.  while (true)
4.  {
5.          int n;
6.   cin >> n;
7.          if (n < 0)
8.          { cout << "negative" << endl;
9.                                  } else
10.  cout << "positive" << endl;
11.         if (n == 0) break;  }
12.     cout << "done loop" << endl;
13. }
```

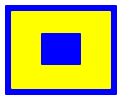
1. How easy do you find this code to read?
2. To the compiler, this is identical to the previous example.

Input and Output

- Every program can read from stdin and write to stdout and stderr
- Typically these correspond to the keyboard and the screen



1. stdin (standard input), stdout (standard output), stderr (standard error).
2. Both stdout and stderr usually go to the screen, but they can be redirected to a file or to other processes, either together or separately.
3. stdin usually comes from the keyboard, but can be redirected to come from a file or the output of another proces.
4. By convention, stdout is used for "normal" output, and stderr for "error" output, but the meaning of these two is not always obvious.
5. This kind of console IO is becoming less and less important as graphical user interfaces become more common.



Streams

- Let you read from `stdin` and write to `stdout`
- Let you read/write files
- Let you read/write `strings` in memory
- Are extensible
- For now, we will just look at writing to `stdout`

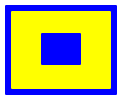
1. In C++, three file descriptors are available by default, as in any program in Unix: stdin, stdout and stderr. Typically stdin comes from the console keyboard, stdout and stderr are both printed to the console display.
2. The console is the display of the computer the program is run on, or the shell it is run from.
3. In Windows, console programs are run from a DOS shell; in Unix from your shell of choice.

Printing to stdout

- The header file `<iostream>` contains the necessary declarations
- `cout` is an object (global variable) that is connected to standard output
- An operator, <<, is defined for all the built-in types

```
1. cout << "here is a number: " << 666
2.      << " and a character: " << '&'
3.      << " pi to a Texan:" << 4.0
4.      << " newline and flush: " << endl;
```

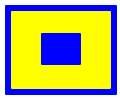
1. So to use cout, you must include <iostream>.
2. Don't worry about what an object is – for now, just think cout as something that is magically created and put in the std namespace.
3. Import the standard library to use it (we will see other options later), with the instruction using namespace std;
4. String literals are enclosed in double quotes: "string", character literals in single quotes: 'c'.
5. There is a (probably apocryphal) story that at some time (in the mid 19th century?) the state of Texas in the USA passed legislation requiring that children be taught that the mathematical constant pi was equal to 4 [sic].
6. Endl is an object which, when printed, sends a newline to the stream and then forces the stream to flush itself (to actually print out anything that may just be buffered at the moment).
7. The built-in types include integers, characters, and real numbers (floating point).
8. String literals are a bit of an oddity in C++ (they are actually pointers to null-terminated character sequences in memory), but this is a detail you needn't worry about at the moment. We will cover pointers and how they relate to arrays in chapter XXX.



Development Environments

- Editor
- Compiler
- Debugger
- Class browser
- Profiler
- Version management

1. There are many development environments available. At the moment, the trend is to use Microsoft Visual Studio for C++ programming in industry.
2. Borland Builder (actually, Borland is now called Inprise, but many people refer to this as borland builder) is very popular; it has a Visual Component Library (VCL) that makes many gui components available to the developer in a graphical view.
3. CodeWarrior is quite popular.
4. KAI is a commercial compiler I don't know much about.



Editors

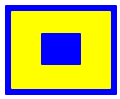
- A good editor provides
  - indentation facilities
  - syntax coloring
  - matching parentheses and braces
  - integration with the compiler
  - integration with the debugger

1. Show examples of what editors look like
2. emacs, VC, Borland, CodeWarrior

Compiler

- There are many C++ compilers
  - Unix
    - SGI, Sun, IBM, HP, g++, egcs
  - Windows
    - VC++, BCB, KAI, Comeau
  - Macintosh
    - ??
- Important aspects
  - Conformity
  - Efficiency of code

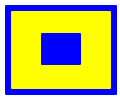
1. Silicon Graphics, International Business Machines, Hewlett-Packard.
2. Visual C++, Borland C++ Builder, KAI?
3. Comeau?
4. Online compiler
5. Latest versions of each



Debugger

- Most Integrated Development Environments have a built-in debugger
- Debuggers typically let you
  - set breakpoints
  - step through the program
  - show data
  - edit data
  - watch variables

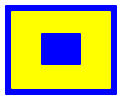
1. Feature table for debuggers
2. VC, Borland, gdb, Sun?



Class Browser

- Shows a hierarchy of classes graphically
- Lets you navigate around
- Lets you see more or less detail for each class

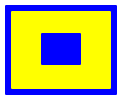
1. What do VC and Borland have?
2. What good ones are commercially available?
3. Rose, other round-trip tools
4. Show examples



Profilers

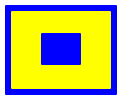
- Let you see where time is being spent
- Graphical profilers are very helpful
- Quantify

1. Mention gprof and prof, and more recent versions



Version Management

- SCCS, RCS, CVS
- SourceSafe, Source OffSite
- Others?



Bug Tracking

- <need material for this>