A starting example

• A program that:
  – Writes a prompt to enter your name
  – Reads the name
  – Writes out “Hello <name>”

• code/first_example/
  – Build it with c++ main.cpp

• What happens if:
  – You don't give any name?
  – You give more than one name?
```cpp
#include <iostream>
#include <string>

int main()
{
    std::cout << "What's your name? 
    std::string name;
    std::cin >> name;
    std::cout << "Hello " << name << 'n';
}
```

- `#include <iostream>` and `#include <string>` import standard input/output facilities and standard string facilities.
- `cout` represents the output `<<` is the output operator.
- `cin` represents the input `>>` is the input operator.
- A statement usually ends with a `;`.
- `\n` is the "newline" character.
- Define a variable of string type.
• Standard facilities are imported with an appropriate `#include` directive
  – Typically they correspond to a file
• The entry point in a program is a function called `main`
  – One possible form of `main` does not take any argument and returns an integer
• A variable must be declared before being used
• `cin` and `cout` represent (standard) input and output respectively
• Operators >> and << are used to read from input and to write to output
• Multiple reads and multiple writes can be chained one after the other
• A statement usually ends with a semi-colon (;)
Identifiers

- An identifier is a user-defined name that denotes program entities (variable, functions, classes, namespaces...)
- An identifier is composed of one or more characters
  - The first character must be a letter (letters include the underscore '_')
    - C++ is case sensitive
  - The other characters can be letters or numbers
  - The length is unlimited (system-specific)
Identifiers (2)

- C++ keywords and alternative tokens cannot be used as identifiers
- Names that:
  - start with an underscore and a capital letter or
  - contain two consecutive underscores are reserved
# C++ keywords

<table>
<thead>
<tr>
<th>asm</th>
<th>do</th>
<th>if</th>
<th>return</th>
<th>typedef</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
<td>double</td>
<td>inline</td>
<td>short</td>
<td>typeid</td>
</tr>
<tr>
<td>bool</td>
<td>dynamic_cast</td>
<td>int</td>
<td>signed</td>
<td>typename</td>
</tr>
<tr>
<td>break</td>
<td>else</td>
<td>long</td>
<td>sizeof</td>
<td>union</td>
</tr>
<tr>
<td>case</td>
<td>enum</td>
<td>mutable</td>
<td>static</td>
<td>unsigned</td>
</tr>
<tr>
<td>catch</td>
<td>explicit</td>
<td>namespace</td>
<td>static_cast</td>
<td>using</td>
</tr>
<tr>
<td>char</td>
<td>export</td>
<td>new</td>
<td>struct</td>
<td>virtual</td>
</tr>
<tr>
<td>class</td>
<td>extern</td>
<td>operator</td>
<td>switch</td>
<td>void</td>
</tr>
<tr>
<td>const</td>
<td>false</td>
<td>private</td>
<td>template</td>
<td>volatile</td>
</tr>
<tr>
<td>const_cast</td>
<td>float</td>
<td>protected</td>
<td>this</td>
<td>wchar_t</td>
</tr>
<tr>
<td>continue</td>
<td>for</td>
<td>public</td>
<td>throw</td>
<td>while</td>
</tr>
<tr>
<td>default</td>
<td>friend</td>
<td>register</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td>goto</td>
<td>reinterpret_cast</td>
<td>try</td>
<td></td>
</tr>
</tbody>
</table>

## C++ alternative tokens

<table>
<thead>
<tr>
<th>and</th>
<th>and_eq</th>
<th>bitand</th>
<th>bitor</th>
<th>compl</th>
<th>not</th>
</tr>
</thead>
<tbody>
<tr>
<td>not_eq</td>
<td>or</td>
<td>or_eq</td>
<td>xor</td>
<td>xor_eq</td>
<td></td>
</tr>
</tbody>
</table>
Another example

• code/second_example/
• What does this program do?
• Try it
  – Build it with c++ main.cpp
• What happens if there is no strings.txt?
  – What happens if there is no test?
• Read from standard input rather than from file
• Read strings rather than whole lines
• Read integers rather than strings
```
#include <vector>
#include <string>
#include <fstream>
#include <algorithm>  // for sort
#include <iostream>

int main()
{
    std::ifstream input_file("strings.txt");
    if (!input_file) {
        std::cerr << "Cannot open file strings.txt\n"
                    return 1;
    }

    std::vector<std::string> v;
    std::string line;
    while (getline(input_file, line)) {
        v.push_back(line);
    }
    sort(v.begin(), v.end());
    for (int i = 0; i < v.size(); ++i) {
        std::cout << v[i] << '\n';
    }
}
```

- Import standard vector facilities
- Import standard file I/O facilities
- Import some standard algorithms
- C++-style comment
- Open strings.txt for reading
- Check if the file was correctly opened
- cerr represents the output for errors
- Exit from main (and from the program) with status 1
- We keep the strings in a vector
- Read a line at a time from the input file
- Append it to the vector
- Sort the vector (in place)
- Print all the strings
- Implicitly return 0 (valid only for main)
Some abstractions...

- Procedural abstraction, e.g. getline, sort
- Data abstraction, e.g. vector, string, ifstream
- Type abstraction, e.g. getline, sort, vector, iterators, string, ifstream
  - Abstractions can be combined
• The C++ library offers a rich set of predefined algorithms
• A C++-style comment starts at the token // and goes until the end of the line
• C-style comments are also supported
  - The comment is enclosed between the tokens /* and */, also on multiple lines
• A variable can be declared just when it's needed
  - This is recommended
• An expression can be automatically converted to a boolean value (true/false), e.g. when testing for if, while and for conditions
  - The conversion can be built-in or user-defined
• The operator ! (not) negates a boolean value
• ++ is the increment operator
  - The expression ++i increments i by 1
• \textit{vector} is a standard container
  – Others exist: \textit{list, map, set}...

• \texttt{[]} is the subscript operator
  – \texttt{v[i]} accesses the \textit{i}^{\text{th}} element of \texttt{v}

• \texttt{begin()} and \texttt{end()} denote the first and one beyond the last element of the vector
  – The denoted range is half-open
Objects, variables, types

- An object is a region of storage
- An object is created by a definition
  - Or by other means we'll see later
- The properties of an object (i.e. its type) are determined when the object is created
  - A type defines the proper use of an object
- A variable is introduced by the declaration of an object. The variable's name denotes the object
Built-in types

- A boolean type (*bool*)
  - *true*, *false*

- Character types (*char* and *wchar_t*; *signed*, *unsigned*)
  - 'a', '5', '\n', '\t'

- Integer types (*short*, *int*, *long*; *signed*, *unsigned*)
  - 234, -7483, 0456 (octal), Oxdead (exa), 878U (force unsigned), 8475L (force long)

- The above are integral types
Built-in types

- Floating-point types (*float*, *double*, *long double*)
  - 1., 1.0, -2.4637, .7e-2, .7F (force float), .7L (force long double)

- Integral types and floating-point types are arithmetic types

```c
int i;        // definition; its value may be undefined
int i = 0;    // definition; initialized to zero
int i(0);     // definition; initialized to zero
int i = int(); // definition; initialized to zero
extern int i; // declaration only
double d;     // definition; its value may be undefined
double d = 3.; // definition; initialized to 3.
char c = 'f'; // definition; initialized to 'f'
bool b = true; // definition; initialized to true
```
Built-in types

• void
  – Special type to mean that no type information is available
    • e.g. No return value from a function

```c
void v;     // error
void f();   // ok, function declaration with no return value
```
User-defined types

• It's possible to combine built-in types to construct other, user-defined types
  - Pointers
  - Arrays
  - References
  - Data structures and classes

• User-defined types can themselves be the basis for further aggregations
Expressions

• An expression is a sequence of operators and operands that specifies a computation

• An expression can result in a value and cause side-effects

• Expression operands are variables and literals

• Appropriate conversions (build-in and user-defined) are executed to adjust the type of the operands
• The order of evaluation of subexpressions within an expression is undefined

• The precedence of operators is the “usual one”
  - Arithmetic > logical > assignment
  - If in doubt, use parenthesis
(Some) Operators

- Arithmetic
  
  \[\text{++ - - + - * / %}\]

- Binary
  
  \[\text{~ | & ^ >> <<}\]

- Logical
  
  \[\text{!! && ||}\]

- Comparison
  
  \[== != < > >= <=\]

- Assignment
  
  \[= += -= *= /= %= <<= >>= &= |= ^=\]

- Subscript
  
  \[[]\]

- Conditional expr
  
  \[?:\]

- Function call
  
  \[(\)]

- Scope resolution
  
  \[::\]
Statements

- Label
- Expression
- Compound (block)
- Selection
  - if, switch
- Iteration
  - while, do-while, for
- Jump
  - break, continue, goto
- Declaration
- Try block
Expression statement

Typically `<expression>` is an assignment or a function call

```java
y = x + 1;
f(x);
```

The result of the expression is thrown away

`<expression>` can be missing (empty statement)
 Compound statement

\[
\{ \text{<statement list> } \}
\]

• So that several statements can be used where one is expected
  – Function body, \textit{if} block, \textit{while} block...

• A compound statement defines a local scope
if (selection statement)

if (<condition>) <statement>
if (<condition>) <statement> else <statement>

- If the condition is true the first statement is executed
- If the condition is false the second statement, if present, is executed
- Condition must be convertible to bool

if (error) {
    std::cerr << “An error occurred\n”;
}

if (x > y) {
    max = x;
} else {
    max = y;
}
**switch** (selection statement)

```
switch ( <condition> ) <statement>
```

- But typically `<statement>` is a sequence of `case` statements and an optional `default` statement

```cpp
switch (n) {
    case 1:
        std::cout << "one\n";
        break; // without this it would continue through case 2
    case 2:
        std::cout << "two\n";
        break;
    default:
        std::cout << "unknown\n"
}
```
Switch and enum

- The `switch` statement and an `enum` type work well together

```cpp
enum rgb { red, green, blue };  
rgb color;                 
switch (color) {
    case red:              
        std::cout << "red\n"; 
        break;                
    case green:            
        std::cout << "green\n"; 
        break;                
    case blue:             
        std::cout << "blue\n"   
        break;                
}
```

The compiler warns if some cases are left out
Iteration statements

```plaintext
while (<condition>) <statement>
```

- `<statement>` is executed as far as `<condition>` is true
  - `<statement>` is executed zero or more times

```plaintext
do <statement> while (<expression>)
```

- `<statement>` is executed as far as `<expression>` is true
  - `<statement>` is executed one or more times
  - `<expression>` must be convertible to `bool`
for (iteration statement)

for (<initialization>; <condition>; <expression>) <statement>

- Roughly equivalent to

```java
{
    <initialization>;
    while (<condition>) {
        <statement>
        <expression>;
    }
}
```

- A for loop is preferred to a while loop when there is a variable controlling the loop
Jump statements

- Exits from an iteration or from a switch
  ```
  break;
  ```

- Terminates the current iteration of a loop
  ```
  continue;
  ```

- Returns from a function with a return value equal to the result of `<expression>`
  ```
  return <expression>;
  ```

- Jump the execution to `<label>`

- ARE YOU SURE YOU WANT TO USE IT?
Functions

• A function is the C++ mechanism to support code abstraction
  – Inherited from C

• The function internal workings are hidden to its clients

• A function may take one or more input parameters

• A function may return a value

• A function may have side-effects
  – Changes to entities not explicitly mentioned in its parameter list
```cpp
#include <vector>
#include <string>
#include <algorithm>
#include <iostream>

std::vector<std::string> read();
void write(std::vector<std::string> v) {
    for (unsigned int i = 0; i < v.size(); ++i) {
        std::cout << v[i] << '\n';
    }
}

int main() {
    std::vector<std::string> v = read();
    sort(v.begin(), v.end());
    write(v);
}

std::vector<std::string> read() {
    std::vector<std::string> result;
    std::string line;
    while (getline(std::cin, line)) {
        result.push_back(line);
    }
    return result;
}
```
• As for variables, a function must be declared before its use

• A function declaration is a definition if it specifies also the function body

• The client calls a function passing appropriate parameters in the right order
  – Formal parameters are initialized with the actual parameters

• The client can ignore the return value

• Reading from cin and writing to cout are examples of side effects
• Functions can be overloaded
  – Same name, different number and/or type of parameters
  – The compiler will choose the best match for a call
  – The return type is not involved

```cpp
std::vector<int> read();
std::vector<int> read(int max_numbers);
std::list<int> read(); // error
```

• Function parameters can have default values

```cpp
std::vector<int> read(int max_numbers = -1);
void write(std::vector<int> v = std::vector<int>());
```
Pass by value vs Pass by reference

- The body of the function `write()` works on a copy of its parameter, i.e. a copy of the vector of strings is made before calling `write()`
  - No changes visible to the client
- If changes need to be visible to the client or the parameter is not copyable or the parameter is “big”, another mechanism is available (passing a “reference” to the original)
References

- A reference declaration introduces an alternative name for an already declared object
- Must be initialized
- Cannot be changed after initialization
  - always bound to the initial object

```c
int i;
int& ri;    // error
int& ri = i; // ok
int j;
ri = j;     // assigns j to i!
int& rc = 1; // error, otherwise we could modify a constant
int const& rc = 1; // ok, a temporary int is created
```
How to pass parameters to functions

• Rule of thumb (there are exceptions)
• If changes need not be visible to the caller:
  – Pass built-in types by value
  – Pass non-built in types by const reference
• If changes need to be visible to the caller
  – Pass always by reference
How to structure source code

• Usually it is convenient to split source code in more than one file, possibly in more than one directory
  – C++ supports separate compilation of semi-independent modules

• Usually it is convenient to factor out some functionality into a reusable “library”

• Usually it is convenient to “export” the interface of the abstraction without showing the actual implementation
Header and source files

• Given a certain functionality:
  – Its interface goes into a “header file”
  – Its implementation goes into a “source file”

• Clients of that functionality will include the header file

• To ensure consistency, the source file includes the header file as well
Data abstraction

• Hide internal representation of an object and allow its manipulation only via its public interface

• Let's implement an abstraction for a complex number

• Start from how we want to use that abstraction
  - Creation of numbers
  - Manipulation
  - Assignments
  - Operations
By default class members are private

The `class` keyword can be replaced by `struct`
- By default class members are public
Complex c1;     // (0.0,0.0)
Complex c2(1.0); // (1.0,0.0)
Complex c3(1.0, 2.0); // (1.0,2.0)
Complex c4(c3);  // (1.0,2.0)
Complex c5 = c3;  // (1.0,2.0)

Creation

c1 = c2;
c1 = 1.0;

Assignment

c1 + c2;
c1 - c2;
c1 * c2;
c1 / c2;
sqrt(c1);

Operations

c1 += c2;
c1 = c2 / c3 + c4;
write(c1);
c1 = read();
c1 = sqrt(c1) * (c2 + c3);

And also...
Class constructor

• A constructor is a special member function that has the same name as the class
  - No return value
• It is invoked at object creation time to initialize the storage allocated to the object
• It can be overloaded
  - And often it is
Special forms of ctors

• Default
  - No parameters
  - Automatically generated if no other ctor is available

• Copy ctor
  - Takes a const reference to another instance of the same class
  - Used to initialize an object as a copy of an existing one
  - Automatically generated if not defined

Complex c1;
Complex c1 = Complex();

Complex c1;
Complex c2(c1);
Complex c2 = c1;
Complex c2 = Complex();
Destructor

• A destructor is a special member function that has the same name as the class prefixed with the character ~
  - No return value
  - No parameters

• It is invoked when the object is about to disappear to release possible resources owned by the object
  - Dynamic memory, files, locks, sockets...

• Automatically generated if not defined
Assignment operator

• Another special member function is the assignment operator (operator=())
  - Takes a const reference to another instance of the same class
  - It should return a reference to itself to allow chaining of assignments

• Used to assign an object to another one (already existing)

```cpp
Complex c1; Complex c2;
c1 = c2;
```

• Automatically generated if not defined
User-defined conversion

- One-parameter ctors can be used for user-defined conversions

```cpp
void f(Complex c);
class Complex {
    Complex(double);
};
Complex c(1.);  // calls Complex(double);
Complex c = 1.;  // equivalent to Complex c = Complex(1.);
f(1.);          // equivalent to f(Complex(1.));
```

- An `explicit` ctor disables this possibility

```cpp
void f(Complex c);
class Complex {
    explicit Complex(double);
};
Complex c(1.);  // calls Complex(double);
Complex c = 1.;  // error
f(1.);          // error
```
Binary operators

- Binary operators such as `+`, `-`, `*`, `/` should be implemented as free functions to allow conversion for both operands.

```cpp
class Complex {
    Complex operator+(Complex const& rhs);
};
Complex c1, c2;
c1 + c2; // ok, calls c1.operator+(c2);
c1 + 1.; // ok, calls c1.operator+(Complex(1.));
1. + c2; // error, no conversion on the first operand
```

```cpp
class Complex { /* ... */ };;
Complex operator+(Complex const& lhs, Complex const& rhs);
Complex c1, c2;
c1 + c2; // ok, calls operator+(c1, c2);
c1 + 1.; // ok, calls operator+(c1, Complex(1.));
1. + c2; // ok, calls operator+(Complex(1.), c2);
```
• Binary operators such as + - * / produce new objects
  – To be returned by value
• Binary operators such as + - * / should be implemented in terms of += -= *= /=
  – Code reuse
• operator++
Include guard

- If the header file is not adequately protected multiple definitions can happen (violation of ODR)

```cpp
#include "complex.hpp"
#include "complex.hpp" // causes compilation error
  // class Complex is defined twice
```

- All header files should be written as follows:

```cpp
#ifndef COMPLEX_HPP
#define COMPLEX_HPP
  // previous stuff here
#endif
```
Namespaces

• Namespaces are a mechanism to partition the space of names in a C++ program
  - The same name can be chosen in different places (possibly by different people, parties, vendors)

```
// complex.hpp
namespace math {
  class Complex { /* usual stuff here */ };  
  Complex operator+(Complex const&, Complex const&);  
  //...
}
```
• Argument Dependent Lookup
  - aka Koenig lookup
  - Unqualified functions/operators are looked for also in the argument's namespace(s)

```cpp
math::Complex c1;
math::Complex c2;
c1 + c2; // calls math::operator+(c1, c2)
```

• Using directive
  - “imports” specified symbol

```cpp
using math::Complex;
Complex c;
```

• Using declaration
  - “imports” all the symbols in the specified namespace

```cpp
using namespace math;
Complex c;
```
• Namespaces can be nested

```cpp
namespace math {
    namespace advanced {
        class Complex { /* usual stuff here */ };  
        Complex operator+(Complex const&, Complex const&);  
        // ...
    }
}
namespace ma = math::advanced; // namespace alias
ma::Complex

• Anonymous namespace
  - Guarantees the uniqueness of the namespace name in the TU where that code ends up

```cpp
namespace {
    void f();
}
f();
```

```cpp
namespace <some unique name> {
    void f();
}
using namespace <some unique name>
f();
```
Dynamic memory

- Sometimes it is useful to create objects that could survive the current scope
- Such objects are created with the `new` operator
- Such objects need to be explicitly destroyed with the `delete` operator

```cpp
class X { /* ... */ };  
X* make_X() { return new X; } 
void f()
{
    X* pointer_to_x = make_X();
    // use pointer_to_x here  
    delete pointer_to_x;
}
```
• The return value of the \textit{new} expression applied to type $T$ is a pointer to $T$ and is denoted by $T^*$

• The use of pointers is very error-prone

\begin{verbatim}
X* p = new X;
p = new X; // ops, how can I delete now the first X?

X* p = new X;
// forgot to delete; memory leak

X* p = new X;
delete p;
delete p; // ops; double delete

X* p = 0;
delete p; // ok, this is valid and does nothing
\end{verbatim}
# How to use pointers

```cpp
class X {}
int main()
{
  X* p1 = new X;
  X x;
  X* p2 = &x;
  delete p1;
}
```

- `operator&()` (address-of) takes the address of an object
class X { int i; };  
X* p = new X;  
X x = *p;  
x.i;  
p->i;  
(*p).i;  
X& x2 = *p;  
X const& x3 = *p;

• . (dot) is the member access operator
  - Cannot be overloaded for user-defined types

• * and -> are the pointer dereference operators
  - (*p).i is equivalent to p->i
  - Can be overloaded (and they are!) for user-defined types