C++

To quote Bjarne Stroustrup:

“C++ is a general-purpose programming language with a bias towards systems programming that:

▶ is a better C
▶ supports data abstraction
▶ supports object-oriented programming
▶ supports generic programming.”

C++

fundamental types

► C++ has all the fundamental types C has
  ► character literals (e.g. 'a') are now of type char
► In addition, C++ defines a new fundamental type, bool
► A bool has two values: true and false
► When cast to an integer, true→1 and false→0
► When casting from an integer, non-zero values become true and false otherwise

enumeration

► Unlike C, C++ enumerations define a new type; for example enum flag {is_keyword=1, is_static=2, isExtern=4, ...}
► When defining storage for an instance of an enumeration, you use its name; for example: flag f = is_keyword
► Implicit type conversion is not allowed:
  f = 5; //wrong    f = flag(5); //right
► The maximum valid value of an enumeration is the enumeration's largest value rounded up to the nearest larger binary power minus one
► The minimum valid value of an enumeration with no negative values is zero
► The minimum valid value of an enumeration with negative values is the nearest least negative binary power
C++ supports references, which provide an alternative name for a variable. Generally used for specifying parameters to functions and return values as well as overloaded operators (more later). A reference is declared with the & operator; for example:

```cpp
int i[] = {1,2}; int &refi = i[0];
```

A reference must be initialised when it is defined. A variable referred to by a reference cannot be changed after it is initialised; for example:

```cpp
refi++; //increments value referenced
```

When used as a function parameter, a referenced value is not copied; for example:

```cpp
void inc(int& i) { i++; } //bad style?
```

Declare a reference as const when no modification takes place. It can be noticeably more efficient to pass a large struct by reference. Implicit type conversion into a temporary takes place for a const reference but results in an error otherwise; for example:

```cpp
float fun1(float&); float fun2(const float&);
```

```cpp
void test() {
    double v=3.141592654;
    fun1(v); //Wrong
    fun2(v);
}
```

Functions doing different things should have different names. It is possible (and sometimes sensible!) to define two functions with the same name. Functions sharing a name must differ in argument types. Type conversion is used to find the “best” match. A best match may not always be possible:

```cpp
void f(double);
void f(long);
void test() {
    f(1L); //f(long)
    f(1.0); //f(double)
    f(1); //Wrong: f(long(1)) or f(double(1)) ?
```

Functions in different scopes are not overloaded; for example:

```cpp
void f(int);
void example() {
    void f(double);
    f(1); //calls f(double);
}
```
Default function arguments

- A function can have default arguments; for example:
  
  ```
  double log(double v, double base=10.0);
  ```

- A non-default argument cannot come after a default; for example:
  
  ```
  double log(double base=10.0, double v); //wrong
  ```

- A declaration does not need to name the variable; for example:
  
  ```
  double log(double v, double=10.0);
  ```

- Be careful of the interaction between `*` and `=`; for example:
  
  ```
  void f(char*=0); //Wrong '*=' is assignment
  ```

Using namespaces

- A namespace is a scope and expresses logical program structure
- It provides a way of collecting together related pieces of code
- A namespace without a name limits the scope of variables to the local execution unit
- The same namespace can be declared in several source files
- The global function `main()` cannot be inside a namespace
- The use of a variable or function name from a different namespace must be qualified with the appropriate namespace(s)
  - The keyword `using` allows this qualification to be stated once, thereby shortening names
  - Can also be used to generate a hybrid namespace
  - `typedef` can be used: `typedef Some::Thing thing;`
- A namespace can be defined more than once
  - Allows, for example, internal and external library definitions

Namespaces

Related data can be grouped together in a `namespace`:

```
namespace Stack { //interface in header
  void push(char);
  char pop();
}
```

```
namespace Stack { //implementation
  const int max_size = 100;
  char s[max_size];
  int top = 0;

  void push(char c) {
    ...
  }
  char pop() {
    ...
  }
}
```

Example

```
namespace Module1 { int x; }
namespace Module2 {
  inline int sqr(const int& i) { return i*i; }
  inline int halve(const int& i) { return i/2; }
}

using namespace Module1; //"import" everything

int main() {
  using Module2::halve; //"import" the halve function
  x = halve(x);
  sqr(x); //Wrong
}
```
Linking C and C++ code

- The directive `extern "C"` specifies that the following declaration or definition should be linked as C, not C++ code:
  ```cpp
  extern "C" int f();
  ```
- Multiple declarations and definitions can be grouped in curly brackets:
  ```cpp
  extern "C" {
      int globalvar; //definition
      int f();
      void g(int);
  }
  ```

User-defined types

- C++ provides a means of defining classes and instantiating objects
- Classes contain both data storage and functions which operate on storage
- Classes have access control:
  - `private`
  - `protected`
  - `public`
- Classes are created with `class` or `struct` keywords
  - `struct` members default to `public` access; `class` to `private`
- A member function with the same name as a class is called a `constructor`
- A member function with the same name as the class, prefixed with a tilde (`~`), is called a `destructor`
- A constructor can be overloaded to provide multiple instantiation methods
- Can create `static` (i.e. per `class`) member variables

Example

```
class Complex {
    double re, im;
public:
    Complex(double r=0.0L, double i=0.0L);
};
Complex::Complex(double r, double i) {
    re=r, im=i;
}
int main(void) {
    Complex c(2.0), d(), e(1.5L);
    return 0;
}
```
Constructors and destructors

- A default constructor is a function with no arguments (or only default arguments)
- If no constructor is specified, the compiler will generate one
- The programmer can specify one or more constructors
- Only one constructor is called when an object is created
- There can only be one destructor
- This is called when an object goes out of scope and is deallocated; this even occurs during exception handling (more later)

Copy constructor

- A new class instance can be defined by assignment; for example:
  ```cpp
  Complex c(1,2);
  Complex d = c;
  ```
- In this case, the new class is initialised with copies of all the existing class' non-static member variables; no constructor is called
- This behaviour may not always be desirable (e.g. consider a class with a pointer as a member variable)
- In which case, define an alternative copy constructor:
  ```cpp
  Complex::Complex(const Complex&)
  ```
- If a copy constructor is not appropriate, make the copy constructor a private member function

Assignment operator

- By default a class is copied on assignment by over-writing all non-static member variables; for example:
  ```cpp
  Complex c(), d(1.0,2.3);
  c = d; //assignment
  ```
- This behaviour may also not be desirable
- The assignment operator (operator=) can be defined explicitly:
  ```cpp
  Complex& Complex::operator=(const Complex& c) {
  ...
  }
  ```

Constant member functions

- Member functions can be declared const
- Prevents object members being modified by the function:
  ```cpp
  double Complex::real() const {
    return re;
  }
  ```