Part 1 — From C to C++

Introduction

Why C++ when we already have C?

C pros:

- Compilers are **FAST**, code is **FAST** (often only a little slower than hand-written assembly)
- Lingua Franca of computing --- C is ubiquitous
- Portability, with C compilers available on all systems
- Compilers/interpreters for new languages are often written in C (Python, Java, JVM, etc.)

Some C issues C++ rectifies:

- Struct initialization and freeing resources is error prone
  - Can't forget to call freeing functions
  - Not always obvious who and when these functions should be called
- Weak support for generic programming (macros / `void*`)
- Object oriented programming not well supported
- Scoped names, helps alleviate name clashing in the global namespace
- Many more!

C vs C++

C can be considered a subset of C++, with only a few exceptions:

- C is more *weakly-typed* regarding pointers

```
void *ptr;
int *i = ptr;   // Implicit conversion from void* to int*
int *j = malloc(5 * sizeof(*i));   // Same as above
int *k = (int *)malloc(5 * sizeof(*i));   // Valid!
```

- C allows for implicit conversion from int to enums

```
typedef enum {Jan, Feb} Month;
Month m1 = 1;   // Invalid conversion from int to Month
Month m2 = static_cast<Month>(1);   // Valid!
```

C vs C++

C++ Additions:

- Reference types, const correctness
- Default parameter values
- Classes, inheritance
- Operator overloading
- Templates, exceptions, namespaces
- Many more!

These make it much easier to manage large projects. Code can be made safer and more readable without sacrificing speed.

C++ implements the *zero-overhead* principle:

You don't pay for features you don't use.

C vs C++

```c
// This is a C program
#include <stdio.h>

int main() {
    printf("Hello world\n");
    return 0;
}
```

```cpp
// This is a C++ program
#include <iostream>

int main() {
    std::cout << "Hello world" << std::endl;
    return 0;
}
```

Basic Building Blocks for C++ Programs

```cpp
// example1.cpp, This is a single line comment
#include <iostream> // preprocessor command: include file
                    // Mostly used for functions, macros,
                    // class definitions.

int foo(int x) {    // Function definition: return type and
                    // parameters block
    return x + 1;   // Return expression value
}

int main() {            // All C/C++ programs start here
    int i = 0;          // Variable declaration
    while (i < 10) {    // Loop
        i = i + 1;      // Expression + assignment
        std::cout << foo(i) << " "; // Operators + call
    }
    if (i >= 10) {      // Conditional
        i = 1;
    } else {
        i = 0;
    }
    return i;           // Return result, exit function
}
```

Basic Building Blocks for C++ Programs

```cpp
// example1.cpp, This is a single line comment
#include <iostream> // preprocessor command: include file
                    // Mostly used for functions, macros,
                    // class definitions.

int foo(int x) {    // Function definition: return type and
                    // parameters block
    return x + 1;   // Return expression value
}

int main() {            // All C/C++ programs start here
    int i = 0;          // Variable declaration
    while (i < 10) {    // Loop
        i = i + 1;      // Expression + assignment
        std::cout << foo(i) << " "; // Operators + call
    }
    if (i >= 10) {      // Conditional
        i = 1;
    } else {
        i = 0;
    }
    return i;           // Return result, exit function
}
```

Basic Building Blocks for C++ Programs

```cpp
// example1.cpp, This is a single line comment
#include <iostream> // preprocessor command: include file
                    // Mostly used for functions, macros,
                    // class definitions.

int foo(int x) {    // Function definition: return type and
                    // parameters block
    return x + 1;   // Return expression value
}

int main() {            // All C/C++ programs start here
    int i = 0;          // Variable declaration
    while (i < 10) {    // Loop
        i = i + 1;      // Expression + assignment
        std::cout << foo(i) << " "; // Operators + call
    }
    if (i >= 10) {      // Conditional
        i = 1;
    } else {
        i = 0;
    }
    return i;           // Return result, exit function
}
```

Basic Building Blocks for C++ Programs

```cpp
// example1.cpp, This is a single line comment
#include <iostream> // preprocessor command: include file
                    // Mostly used for functions, macros,
                    // class definitions.

int foo(int x) {    // Function definition: return type and
                    // parameters block
    return x + 1;   // Return expression value
}

int main() {            // All C/C++ programs start here
    int i = 0;          // Variable declaration
    while (i < 10) {    // Loop
        i = i + 1;      // Expression + assignment
        std::cout << foo(i) << " "; // Operators + call
    }
    if (i >= 10) {      // Conditional
        i = 1;
    } else {
        i = 0;
    }
    return i;           // Return result, exit function
}
```

Basic Building Blocks for C++ Programs

```cpp
// example1.cpp, This is a single line comment
#include <iostream> // preprocessor command: include file
                    // Mostly used for functions, macros,
                    // class definitions.

int foo(int x) {    // Function definition: return type and
                    // parameters block
    return x + 1;   // Return expression value
}

int main() {            // All C/C++ programs start here
    int i = 0;          // Variable declaration
    while (i < 10) {    // Loop
        i = i + 1;      // Expression + assignment
        std::cout << foo(i) << " "; // Operators + call
    }
    if (i >= 10) {      // Conditional
        i = 1;
    } else {
        i = 0;
    }
    return i;           // Return result, exit function
}
```

Basic Building Blocks for C++ Programs

```cpp
// example1.cpp, This is a single line comment
#include <iostream> // preprocessor command: include file
                    // Mostly used for functions, macros,
                    // class definitions.

int foo(int x) {    // Function definition: return type and
                    // parameters block
    return x + 1;   // Return expression value
}

int main() {            // All C/C++ programs start here
    int i = 0;          // Variable declaration
    while (i < 10) {    // Loop
        i = i + 1;      // Expression + assignment
        std::cout << foo(i) << " "; // Operators + call
    }
    if (i >= 10) {      // Conditional
        i = 1;
    } else {
        i = 0;
    }
    return i;           // Return result, exit function
}
```

Basic Building Blocks for C++ Programs

```cpp
// example1.cpp, This is a single line comment
#include <iostream> // preprocessor command: include file
                    // Mostly used for functions, macros,
                    // class definitions.

int foo(int x) {    // Function definition: return type and
                    // parameters block
    return x + 1;   // Return expression value
}

int main() {            // All C/C++ programs start here
    int i = 0;          // Variable declaration
    while (i < 10) {    // Loop
        i = i + 1;      // Expression + assignment
        std::cout << foo(i) << " "; // Operators + call
    }
    if (i >= 10) {      // Conditional
        i = 1;
    } else {
        i = 0;
    }
    return i;           // Return result, exit function
}
```

Comments

It is important to comment your code — for others and yourself!

```
/* This is an old-style
   multi-
   line
   comment (C, also C++)
*/

/**
 * I prefer
 * this style of multi-
 * line comments
 */

// This is a single line comment (C++, but not C!)
```

Multi-line comments cannot be nested: `/* /* */ */`

Comments

Where to put comments?

- Mostly in header files, where functions and structs/classes are defined (this is where the user looks, who often is not interested in implementation)
- At the beginning of files describing their purpose
- On top of function definitions discussing parameters, function effects, and return values
- On top of struct/class definitions describing their purpose
- In front of non-trivial parts, *i.e.*, anything you wouldn't instantly understand when looking at the code a month later!

There is no need to write novels or too comment each program statement!

Namespaces

A *namespace* is a region that provides a named-scope to the identities (functions, variables, user-defined types, etc.)

- Organizes code into logical groups
- prevents name collisions that can occur, especially when including multiple libraries
- ODR (one definition rule!)

Namespaces

Identities at namespace scope are visible to one another without qualification. Identifiers outside the namespace can access members by using the fully qualified name.

The Standard Library uses the namespace `std::`

One can use `using namespace std;` to save typing the fully qualified name, but don't use these in header files!

Namespaces

```cpp
namespace MyLib {
    struct Outer { ... };

    namespace InnerMyLib {
        // Outer level scopes are visible
        struct InnerStruct { Outer my_outer; };
    }

    // Inner scopes are not visible
    void Foo(InnerMyLib::InnerStruct x) { ... }
    void Bar(Outer x) { ... }
}

// Same function name + parameters, but no name collisions!
void Foo(MyLib::InnerMyLib::InnerStruct x) { ... }

using namespace MyLib;
void Bar(Outer x) { ... }
void Foo2(InnerMyLib::InnerStruct x) { ... }
```

Input and Output

Input via input-stream `cin` (standard input)

```
cin >> var1 >> var2 >> ... >> varn;
```

Output via output-stream `cout` (standard output)

```
cout << exp1 << exp2 << ... << expn;
```

```cpp
// example2.cpp
#include <iostream>      // Imports definitions of input/output
                         // stream classes

int main() {
    int n;
    std::cout << "n=?\n";        // Output string
    std::cin >> n;               // Input number and store to n
    std::cout << "2*n=" << (2*n) << std::endl;  // Output string and
                                                // expression result
    return 0;
}
```

Standard Error Stream

Another predefined output stream is `cerr`, which is used for error messages.

```
cerr << "Division by zero" << endl; exit(10);
```

By default, output is also sent to the console, but it is not redirected when using > or |. We can redirect content from the error stream:

```
command > coutfile 2> cerrfile
```

which sends output from `cout` to `coutfile` and output from `cerr` to `cerrfile`.

C/C++ Number Types

C++ uses the same fundamental number types as C: `char`, `short`, `int`, `float`, `double`. C++ adds type `bool` which contains values `true` and `false`.

In C/C++, integer expressions are **NOT** checked for overflows/underflows!

- unsigned arithmetic is defined to *wrap around*
- signed arithmetic over/underflow is undefined behaviour.

C/C++ Number Types

```
unsigned char foo = 255;
unsigned char bar = foo + 1;
// value of bar is 0 because of 2s-complement number representation:
// 255 (base 10) =     1111 1111  (base 2)
//                   + 0000 0001
//                 = (1) 0000 0000 = 0 (base 10)
```

Floating-point overflows/underflows are indicated by special values (`+Inf`, `-Inf`, `NaN`), but the program continues anyways.

C/C++ Number Types

What will the compiler produce for foo and bar?

```
bool foo(int x) {
    return x < x + 1;
}

bool bar(unsigned int x) {
    return x < x + 1;
}
```

TryMe

Const Qualifier

The `const` qualifier makes a variable unchangeable by defining the variable's type as `const`.

```
const int buffer_size = 512;
```

Any attempt to assign to `buffer_size` results in a compiler error. As a result, all `const` variables must be initialized.

We can initialize non-`const` variables to their `const` counterparts, as copying doesn't change that object. Once the object is made, the new object has no further access to the original object.

```
const int buffer_size = 512;
int new_buffer_size = buffer_size;
++new_buffer_size;       // Ok, new_buffer_size is non-const
```

Top-Level and Low-Level Const

```cpp
int i = 0;
int *p = &i;                    // (1)
int *const cp = &i;             // (2) Top-level const
const int *pc = &i;             // (3) Low-level const
const int *const cpc = &i;      // (4) Both top and low-level const
```

1. **Normal pointer**: can be used to change underlying object and can be reassigned
2. **Const pointer**: can be used to change underlying object but cannot be reassigned
3. **Pointer to const**: cannot be used to change underlying object, but can be reassigned
4. **Const pointer to const**: cannot be used to change underlying object and cannot be reassigned

Top-Level and Low-Level Const

Rule: When we copy and object, both objects must have

1. The same low-level `const` qualification, or
2. There must be a conversion between the types of the two objects

In general, we can convert a non-`const` to `const`, but not the other way around

```
int i = 0;
const int *const cpc = &i;

int *p1 = cpc;        // Error: cpc has low-level const, p doesn't
int *const p2 = cpc; // Error: same as above (top-level ignored)
const int *p3 = cpc; // Ok: p3 and cpc share same low-level const
```

> **Tip:** Read the type specifier right-to-left!

Const Correctness

Const-Correctness means that everything that isn't intended to be modified should be marked as `const`. It is a form of type safety, and should be used whenever possible.

- It provides a form of documentation and promise to users that particular variables will not be modified
- It protects you from accidentally changing variables that are not intended to be changed (errors given at compile time!)
- The compiler can generate more efficient code (in some cases)

```
const int x = 1;
const int y = 2;
if (x = y)  // Whoops, should be if(x == y)

const int foo = 2;
int get_foo() {
    return foo;      // Compiler can replace variable foo
}                    // with value 2 directly
```

TryMe

Value Categories

In C++03, an expression is either an *rvalue* or an *lvalue*. C++11 introduces *xvalues*, *glvalues*, and *prvalues*, but its instructive to consider only rvalues and lvalues for now.

- An *lvalue* represents an object that occupies some identifiable location in memory.
  - The name of a variable, function, data member
- An *rvalue* is defined by exclusion, being that every expression is either an lvalue or an rvalue.
  - Literal values (e.g. 42)
  - Function call expressions whose return type is non-reference
  - Cannot take the address of

Value Categories

Example: Assignment operator requies a non-`const` lvalue as its left-hand operand and yields its left-hand operand as an lvalue.

```
const int a = 10;    // 'a' is an lvalue
a = 10;              // Error: non-const lvalue required for assignment.
```

Example: The address-of operator requires an lvalue operand and returns a pointer to its operand as an rvalue.

```
const int a = 10;    // 'a' is an lvalue
int *y = &a;         // & operator takes lvalue and returns rvalue
```

Value Categories

Generally speaking, language constructs operating on object values require rvalue as arguments, and an implicit lvalue-to-rvalue conversion occurs:

```
int a = 1, b = 2;
int c = a + b;       // a and b are converted to rvalues,
                     //  and an rvalue is returned
```

While rvalues can't generally be converted to lvalues, the dereference operator takes an rvalue argument but produces an lvalue as a result:

```
int arr[] = {1, 2, 3};
int *p = &arr[0];
*(p+1) = 10;         // Ok, p+1 is an rvalue but *(p+1) is an lvalue
```

Reference Types

C passes parameters by value (except for arrays). C++ supports parameter references directly, which are *aliases* (alternate names for an object).

- When we define a reference, we *bind* the reference to its initializer
- Once initialized, a reference remains bound to its initial object.
- There is no way to rebind a reference, and all operators performed on that reference are actually operations performed on the object which it is bound to.
- References are not objects, so one cannot define a reference to a reference or a literal (more on this in a bit).

Reference Types

To define a reference, use the & prefix.

```
int A[] = { 1, 2, 3, 4};
int &x = A[2];          // x is a reference bound to A[2]
++x;                    // A is now {1, 2, 4, 4}
```

We can use call-by-reference in functions

```
void increment(int &x) {
    ++x;
}
int y = 5;
increment(y); // y is now 6
```

```
void increment(int x) {
    ++x;
}
int y = 5;
increment(y); // y is still 5
```

Internally, x is a pointer to y which cannot be rebound. We refer to these types of references as `lvalue references`.

References Types

Why and when to use references?

- Similar benefits of using pointers:
  - Functions can modify passed objects
  - Only the address of the object is copied, rather than the whole object (which can be costly)
  - Allows to simulate multiple returned function values
- You access a reference with exactly the same syntax as the name of an object (as opposed to pointers which use `->`)
- There is no `null reference`; thus, we can always assume that a reference refers to a valid

References Types

Example

```
void swap(int &x, int &y) {
    int temp = x; x = y; y = temp;
}
a = 1; b = 2;        // Before: a=1, b=2
swap(a, b);          // After: a=2, b=1
swap(1, 2);          // Error
```

Equivalent C code:

```
void swap(int *px, int *py) {
    int temp = *px; *px = *py; *py = temp;
}
a = 1; b = 2;        // Before: a=1, b=2
swap(&a, &b);        // After: a=2, b=1
```

References to Const

It is often useful to pass constants or results of function calls (rvalues) as arguments to functions with reference parameters:

```
void foo(int &x) {}
foo(10);              // Error, invalid rvalue to lvalue conversion
int x = 10;
foo(x);               // Ok, x is an lvalue
```

While creating a temporary variable works, it can be costly is the object we are copying is large.

A solution to this is to use `const` lvalue references, which allows us to bind `const` lvalues to an rvalue without any temporaries or copies:

```
const int &bar = 10;            // Ok
void foo(const int &x) {}
foo(10)                         // Ok
```

> **Tip:** Declare read-only parameters as const &

Rvalue References

New to C++11 are *rvalue references*, which are denoted by &&. RValue references allows a function to branch at compile time (via overload resolution) on the condition if it is being called on an lvalue or an rvalue.

```cpp
void foo(X &x) {}
void foo(X &&x) {}

X x;
X foobar() {}           // A function which returns an rvalue

foo(x);                 // Argument is lvalue, calls foo(X &)
foo(foobar());          // Argument is rvalue, calls foo(X &&)
```

If we did not distinguish between lvalue and rvalue references, we would have to copy the function argument on invocation.

> **Tip:** Provide lvalue and rvalue overloads for modifiable arguments

Rvalue References

Example:

```
void foo(X &);              // (1)
void foo(const X &);        // (2)
void foo(X &&);             // (3)
```

- If we implement (1) but not (2), then `foo` can be called on lvalues but not rvalues
- If we implement (2) but not (3), then `foo` can be called on both lvalues and rvalues, but we cannot differentiate between lvalues and rvalues.
- If we implement both (2) and (3), then `foo` can be called on both lvalues and rvalues, and we can still differentiate between lvalues and rvalues.
- If we implement (3) but not (1) or (2), then `foo` can only be called on rvalues.

Default Arguments

C++ allows for arguments to have default values. All default arguments must be in the rightmost positions. Omitting arguments begins with the rightmost one.

```
void foo(int a, int b=2, int c=3, int d=4) {}

foo();                          // Illegal
foo(x);                         // calls foo(x, 2, 3, 4)
foo(x, y)                       // calls foo(x, y, 3, 4)
foo(x, y, z)                    // calls foo(x, y, z, 4)
foo(4, 3, 2, 1);                // calls foo(4, 3, 2, 1)

// Illegal:
void bar(int a=1, int b, int c=3, int d);
```

The last example is not allowed as `bar(x, y, z)` would be ambiguous — is c or a assigned a default value?

Dynamic Memory Allocation

Local variables, function parameters, and function call return addresses are located on the runtime *stack*

- Last-in first-out (LIFO) data structure
- Once out of scope, the stack *pops* the data and it is no longer accessible

Dynamic memory that is more permanent and can last out-last function calls is allocated from a different part of memory called *heap*.

- Operator `new` dynamically allocated memory on the heap
- Operator `delete` is used to release it when no longer needed; can be done later, even in a different function

There is no garbage collection in C++; **YOU** are in control because the compiler cannot know when memory is no longer needed and can be deleted.

Operator New

There is no initialization for basic C plain old data (POD) types, unlike Java or Python. Calling new with class types calls the class constructor (more on this later).

Why?

```
int *p = new int;        // Allocated space for one int,
                         // p now points to it
*p = 0;                  // Use the allocted memory
```

Tip: Your code needs to initialize POD explicitly. Otherwise, content is undefined, in which case you can expect random program behaviour!

Operator Delete

Operator `delete` frees the memory its parameter points to.

```
int *p = new int;        // Allocated one int on heap
delete p;                // free memory when integer *p no longer used
```

If `p == 0`, `delete p` does nothing. Before returning the memory back to the operating system, the class destructor for non-POD types is called (more on this later).

> **Tip:** Set pointer to nullptr after delete to prevent further access of this address through this pointer.

```
delete p;
p = nullptr;
```

`nullptr`

In C and C++03, `0` (zero) is a special pointer value that can be assigned to any pointer variable, regardless of type.

`0` is not the address of any process memory, and thus can be used to indicate errors when used as a function return value, or special conditions such as "this linked list node has no successor".

`0` is also an integer constant, which sometimes leads to ambiguities (is it a pointer or is it an integer)? What about the following:

```
void foo(int n);
void foo(char *s);
foo(NULL);                  // Which gets called?
```

In C++11, `nullptr` was introduced to represent a pointer literal. Using C's `NULL`, or `0` pointer value, is discouraged.

New/Delete vs Malloc/Free

So why choose `new`/`delete` over `malloc`/`free`?

- `new` guarantees the calling of constructors of classes for initializing class members, while `malloc` does not (you would need an additional call to initialize)
- If `new` fails to allocate memory, an exception is thrown (more on this later); if no exception is thrown, you can assume that the allocation was successful!
- `malloc` can fail, and so each usage of `malloc` also requires a `NULL` check
- Both `delete` and `free` called twice on the same pointer can lead to undefined behaviour; however calling `delete` on `nullptr` does nothing
- It is undefined behaviour if you mix `malloc` and `delete`, or `new` and `free`

Dynamic Arrays

`new[]` allocates an array of elements of the given type on the heap.

- POD variables are not initialized
- non-POD variables (*i.e.* classes) have their constructors called for each array element

`delete[]` is used to free arrays.

- Before memory is released, `delete` calls destructor for each array element if it is non-POD

```
const int N = 100;
float *p = new float[N];
for (int i = 0; i < N; ++i) {
    p[i] = 0.0;
}
delete[] p;
```

Matching `new` and `delete`

`new` and `delete` come in pairs:

> **Tip:** For every new there should be at least one delete in your program to avoid memory leaks.

More specifically,

- For every `new` at least one corresponding `delete` = For every `new[]` at least one corresponding `delete[]`

If mixed, the computation results is undefined. Such bugs are hard to track. Tools like *valgrind* and setting pointers to `nullptr` after `delete` can help.