ECE 2400 Computer Systems Programming
Fall 2020

Topic 12: Transition to C++

School of Electrical and Computer Engineering
Cornell University

revision: 2020-10-22-19-01

1 Why C++? 3
   1.1. Procedural Programming ............................ 3
   1.2. T13: Object-Oriented Programming .................. 4
   1.3. T14: Generic Programming .......................... 4
   1.4. T15: Functional Programming ......................... 5
   1.5. T16: Concurrent Programming ....................... 6
   1.6. C vs. C++ ........................................... 6

2 C++ Namespaces 7

3 C++ Functions 12

4 C++ References 15

5 C++ Exceptions 19

6 C++ Types 22
   6.1. struct Types ....................................... 22
   6.2. bool Type ......................................... 23
6.3. void* Type .............................................. 23
6.4. nullptr Literal ......................................... 24
6.5. auto Type Inference .................................... 24

7 C++ Range-Based For Loop 25

8 C++ Dynamic Allocation 26
It is absolutely critical for students to take active ownership of their transition from C to C++!

1. Why C++?

• A programming paradigm is a way of thinking about software construction based on some fundamental, defining principles

• Different programming paradigms can potentially enable programmers to tell new stories in possibly more elegant ways

• C supports a limited set of programming paradigms, and this can significantly constrain the kind of stories a programmer can tell

• C++ is (mostly) a superset of C that enables new programming paradigms; C++ is a “multi-paradigm programming language”

• C++ enables programmers to tell richer and more elegant stories

1.1. Procedural Programming

• Programming is organized around defining and calling procedures (i.e., routines, subroutines, functions)

• C primarily supports procedural programming

• Almost all of the C syntax and semantics you have learned so far can be used in C++; thus C++ also supports procedural programming

```c
int avg(int x, int y)
{
    int sum = x + y;
    return sum / 2;
}

int main(void)
{
    int c = avg(2,3);
    return 0;
}
```
1.2. **T13: Object-Oriented Programming**

- Programming is organized around defining, instantiating, and manipulating *objects* which contain data (i.e., fields, attributes) and code (i.e., methods).

- C can (partially) support object-oriented programming through careful policies on using structs and functions.

- C++ adds new syntax and semantics to elegantly support object-oriented programming.

```c
typedef struct
{
    // implementation specific
}
list_int_t;

void list_int_construct ( list_int_t* this );
void list_int_destruct ( list_int_t* this );
void list_int_push_front ( list_int_t* this, int v );
void list_int_reverse ( list_int_t* this );
```

1.3. **T14: Generic Programming**

- Programming is organized around algorithms and data structures where *generic types* are specified upon instantiation as opposed to definition.

- C can (partially) support generic programming through awkward use of the preprocessor and/or `void*` pointers.

- C++ adds new syntax and semantics to elegantly support generic programming.
#define SPECIALIZE_LIST_T( T )

typedef struct
{
    /* implementation specific */
}
list_##T##_t;

void list_##T##_construct ( list_##T##_t* this );
void list_##T##_destruct ( list_##T##_t* this );
void list_##T##_push_front ( list_##T##_t* this, T v );
void list_##T##_reverse ( list_##T##_t* this );

SPECIALIZE_LIST_T( int )
SPECIALIZE_LIST_T( float )

### 1.4. T15: Functional Programming

- Programming is organized around *pure functions* (i.e., function output depends only on the parameters) as first-class primitives that can be manipulated just like other values.

- C can (partially) support functional programming through the use of function pointers.

- C++ adds new syntax and semantics to elegantly support functional programming.

```c
typedef int (*cmp_ptr_t) ( int, int );
int lt( int x, int y ) { return x < y; }

int main( void )
{
    cmp_ptr_t cmp_ptr = &lt;
    int result = (*cmp_ptr)( 3, 4 );
    return 0;
}
```
1.5. T16: Concurrent Programming

• Programming is organized around computations that execute *concurrently* (i.e., computations execute overlapped in time) instead of *sequentially* (i.e., computations execute one at a time)

• C can support concurrent programming through the use of a standard library (e.g., pthreads)

• C++ adds new syntax and semantics to elegantly support concurrent programming

1.6. C vs. C++

• 1972: C development started by Dennis Ritchie and Ken Thompson
• 1979: C++ development started by Bjarne Stroustrup
• C90: First ANSI C standard
• C++98: First ISO C++ standard
• C99: Modern C standard (*this course*)
• C++11: Major C++ revision with many new features (*this course*)
• C++14: Small C++ revision mostly for bug fixes
• C++17: Medium C++ revision with some new features
2. C++ Namespaces

• Large C projects can include tens or hundreds of files and libraries
• Very easy for two files or libraries to define a function with the same name causing a namespace collision

```c
// contents of foo.h
// this avg rounds down
int avg( int x, int y )
{
    int sum = x + y;
    return sum / 2;
}

// contents of bar.h
// this avg rounds up
int avg( int x, int y )
{
    int sum = x + y;
    return (sum + 1) / 2;
}

#include "foo.h"
#include "bar.h"
#include <stdio.h>

int main( void )
{
    printf("avg(2,3) = %d\n", avg(2,3) );
    return 0;
}  
```

• Unclear which version of avg to use
• Causes compile-time “redefinition” error
• Traditional approach in C is to use prefixes
• Can create cumbersome syntactic overhead

```
// contents of foo.h
// this avg rounds down
int foo_avg( int x, int y ) {
    int sum = x + y;
    return sum / 2;
}

// contents of bar.h
// this avg rounds up
int bar_avg( int x, int y ) {
    int sum = x + y;
    return (sum + 1) / 2;
}

#include "foo.h"
#include "bar.h"
#include <stdio.h>

int main( void ) {
    printf("avg(2,3) = %d (rnd down)\n", foo_avg(2,3) );
    printf("avg(2,3) = %d (rnd up)\n", bar_avg(2,3) );
    return 0;
}
```

https://repl.it/@cbatten/ece2400-T12-ex2
• **C++ namespaces** extend the language to support named scopes
• Namespaces provide flexible ways to use specific scopes

```cpp
// contents of foo.h
namespace foo {

    // this avg rounds down
    int avg( int x, int y )
    {
        int sum = x + y;
        return sum / 2;
    }

    // Other code in
    // namespace uses "avg"
}

#include "foo.h"
```

```cpp
// contents of bar.h
namespace bar {

    // this avg rounds up
    int avg( int x, int y )
    {
        int sum = x + y;
        return (sum + 1) / 2;
    }

    // Other code in
    // namespace uses "avg"
}
```

```cpp
#include "bar.h"
#include <stdio.h>

int main( void )
{
    printf("avg(2,3) = %d (rnd down)\n", foo::avg(2,3) );
    printf("avg(2,3) = %d (rnd up)\n", bar::avg(2,3) );
    return 0;
}
```

```cpp
int main( void )
{
    using namespace foo;
    printf("avg(2,3) = %d (rnd down)\n", avg(2,3) );
    printf("avg(2,3) = %d (rnd up)\n", bar::avg(2,3) );
    return 0;
}
```

https://repl.it/@cbatten/ece2400-T12-ex3
• Namespaces are just syntactic sugar
• Useful way to group related struct and function definitions

```cpp
namespace ListInt {
    typedef struct _node_t {
        int value;
        struct _node_t* next_p;
    } node_t;
}

namespace ListInt {
    void construct ( list_t* this );
    void destruct ( list_t* this );
    void push_front( list_t* this, int v );
    ...
}

int main( void ) {
    ListInt::list_t list;
    ListInt::construct ( &list );
    ListInt::push_front( &list, 12 );
    ListInt::push_front( &list, 11 );
    ListInt::push_front( &list, 10 );
    ListInt::destruct ( &list );
    return 0;
}
```
2. C++ Namespaces

- Can rename namespaces and import one namespace into another
- All of the C standard library is placed in the std namespace
- Use the C++ version of the C standard library headers

```cpp
#include "foo.h"
#include "bar.h"
#include <cstdio>

int main ( void )
{
    std::printf("avg(2,3) = %d (rnd down)\n", foo::avg(2,3) );
    std::printf("avg(2,3) = %d (rnd up)\n", bar::avg(2,3) );
    return 0;
}
```

- `<assert.h>` <cassert> conditionally compiled macro
- `<errno.h>` <cassert> macro containing last error num
- `<fenv.h>` <cfenv> floating-point access functions
- `<float.h>` <cfloat> limits of float types
- `<inttypes.h>` <cinttypes> formatting macros for int types
- `<limits.h>` <climits> limits of integral types
- `<locale.h>` <clocale> localization utilities
- `<math.h>` <cmath> common mathematics functions
- `<setjmp.h>` <csetjmp> for saving and jumping to execution context
- `<signal.h>` <csignal> signal management
- `<stdarg.h>` <cstdarg> handling variable length arg lists
- `<stddef.h>` <cstddef> standard macros and typedefs
- `<stdint.h>` <cstdint> fixed-size types and limits of other types
- `<stdio.h>` <cstdio> input/output functions
- `<stdlib.h>` <cstdlib> general purpose utilities
- `<string.h>` <cstring> narrow character string handling
- `<time.h>` <ctime> time utilities
- `<ctype.h>` <cctype> types for narrow characters
- `<uchar.h>` <cuchar> unicode character conversions
- `<wchar.h>` <cwchar> wide and multibyte character string handling
- `<wctype.h>` <cwctype> types for wide characters
3. C++ Functions

- C only allows a single definition for any given function name

```c
int avg ( int x, int y );
int avg3( int x, int y, int z );
```

- **C++ function overloading** allows multiple def per function name
- Each definition must have a unique function signature (e.g., number of parameters)

```c
int avg( int x, int y )
{
    int sum = x + y;
    return sum / 2;
}

int avg( int x, int y, int z )
{
    int sum = x + y + z;
    return sum / 3;
}

int main()
{
    // Will call definition of avg with 2 parameters
    int a = avg( 10, 20 );

    // Will call definition of avg with 3 parameters
    int b = avg( 10, 20, 25 );

    return 0;
}
```
• C only allows a single definition for any given function name

```c++
int avg ( int x, int y );
double favg ( double x, double y );
```

• Function overloading also enables multiple definitions with the same number of arguments but different argument types

```c++
int avg ( int x, int y )
{
  int sum = x + y;
  return sum / 2;
}

double avg ( double x, double y )
{
  double sum = x + y;
  return sum / 2;
}

int main()
{
  // Will call definition of avg with int parameters
  int a = avg ( 10, 20 );

  // Will call definition of avg with double parameters
  double b = avg ( 7.5, 20 );

  return 0;
}
```
• **Default parameters** can allow the caller to *optionally* specify specific parameters at the *end* of the parameter list

```cpp
#include <cstdio>

enum round_mode_t {
    ROUND_MODE_FLOOR, 
    ROUND_MODE_CEIL,
};

int avg( int a, int b, 
    round_mode_t round_mode = ROUND_MODE_FLOOR )
{
    int sum = a + b;
    if ( round_mode == ROUND_MODE_CEIL )
        sum += 1;
    return sum / 2;
}

int main( void )
{
    std::printf("avg( 5, 10 ) = %d\n", avg( 5, 10 ));
    return 0;
}
```

• Function overloading and default parameters are just syntactic sugar
• Enable elegantly writing more complicated code, but must also be more careful about which function definition is actually associated with any given function call
4. C++ References

- C provides pointers to indirectly refer to a variable

```c
int a = 3;
int* const b = &a;
*b = 5;
int c = *b;
int* d = &(*b);
```

Aside:

```c
// cannot change value
const int a;

// cannot change pointed-to value
const int* b = &a;

// cannot change pointer
int* const b = &a;

// cannot change pointer or pointed-to value
const int* const b = &a;
```

- Pointer syntax can sometimes be cumbersome (we will see this later with operator overloading)
- C++ references are an alternative way to indirectly refer to variable
- References require introducing new types
- Every type T has a corresponding reference type T&
- A variable of type T& contains a reference to a variable of type T

```c
int& a // reference to a variable of type int
char& b // reference to a variable of type char
float& c // reference to a variable of type float
```
4. C++ References

- Cannot declare and then assign to reference in separate statements
- Must always use an initialization statement
- Do not use address-of operator (&) to initialize reference

```cpp
int a = 42;
int& b;       // reference to a variable (illegal)
b = &a;       // assign to reference (illegal)
int& c = &a;  // initialize ref with address of (illegal)
int& c = a;   // initialize reference (legal)
```

- For the most part, references act like syntactic sugar for pointers
- References must use an initialization statement (cannot be NULL)
- Cannot change reference after initialization
- References are automatically dereferenced
- References are a synonym for the referenced variable

![Stack Diagram]

```
int a = 3;       // int a = 3;
int& b = a;      // int* const b = &a;
b = 5;           // *b = 5;
int c = b;       // int c = *b;
int* d = &b;     // int* d = &(b);
```
• Using pointers for call-by-reference

```cpp
void sort( int* x_ptr,
           int* y_ptr )
{
    if ( (*x_ptr) > (*y_ptr) ) {
        int temp = *x_ptr;
        *x_ptr = *y_ptr;
        *y_ptr = temp;
    }
}
```

• Using references for call-by-reference

```cpp
void sort( int& x, int& y )
{
    if ( x > y ) {
        int temp = x;
        x = y;
        y = temp;
    }
}

int main( void )
{
    int a = 9;
    int b = 5;
    sort( a, b );
    return 0;
}
```

https://repl.it/@cbatten/ece2400-T12-ex5
Draw a state diagram corresponding to the execution of this program

```cpp
void avg( int& result, int x, int y )
{
    int sum = x + y;
    result = sum / 2;
}

int main( void )
{
    int a = 10;
    int b = 20;
    int c;
    avg( c, a, b );
    return 0;
}
```

- Our coding conventions prefer using pointers for return values
- Makes it obvious to caller that the parameter can be changed
- Const references useful for passing in large values

```cpp
void blur( image_t* out, const image_t& in );

int main( void )
{
    image_t in = /* initialize */
    image_t out = /* initialize */
    blur( &out, in );
    return 0;
}
```
5. C++ Exceptions

• When handling errors in C we can return an invalid value

```c
int days_in_month( int month )
{
    int x;
    switch ( month )
    {
        case 1: x = 31; break;
        ...
        case 12: x = 31; break;
        default: x = -1;
    }
    return x;
}
```

• When handling errors in C we can assert

```c
int days_in_month( int month )
{
    assert( (month > 1) && (month <= 12 ) );
    ...
}
```

```c
int main()
{
    int result = days_in_month( 7 );
    // Indicate success to the system
    return 0;
}
```
• C++ exceptions enable global non-linear control flow

```cpp
int x = 1;
try {
    int y = 10;
    if (x)
        throw -1;
    y = 11;
    x = 2;
} catch (int e) {
    x = e;
}
```

```cpp
int x = 0;
try {
    int y = 10;
    if (x)
        throw -1;
    y = 11;
    x = 2;
} catch (int e) {
    x = e;
}
```
5. C++ Exceptions

- C++ exceptions enable cleanly throwing and catching errors

```cpp
#include <cstdio>

int days_in_month( int month )
{
    int x;
    switch ( month )
    {
        case 1: x = 31; break;
        case 2: x = 28; break;
        ...
        case 12: x = 31; break;
        default: 
            throw -1;
        }
    return x;
}

int main()
{
    try {
        int month = 7;
        int days = days_in_month( month );
        std::printf( "month %d has %d days\n", month, days );
    } catch ( int e ) {
        std::printf( "ERROR: %d\n", e );
        return e;
    }

    // Indicate success to the system
    return 0;
}
```

https://repl.it/@cbatten/ece2400-T12-ex6
• Can throw variable of any type (e.g., integers, structs)
• Can catch and rethrow exceptions
• Uncaught exceptions will terminate the program

6. C++ Types

Small changes to two types, one new types, one new literal, and a new way to do type inference

• struct types
• bool type
• void* type

• nullptr literal
• auto type inference

6.1. struct Types

• C++ supports a simpler syntax for declaring struct types

```cpp
typedef struct {
  double real;
  double imag;
} complex_t;

int main( void ) {
  complex_t complex;
  complex.real = 1.5;
  complex.imag = 3.5;
  return 0;
}
```

• C coding convention uses _t suffix for user defined types
• C++ coding convention uses CamelCase for user defined types
6.2. bool Type

- C used int types to represent boolean values
- C++ has an actual bool type which is part of the language
- C++ provides two new literals: true and false
- C++ still accepts integers where a boolean value is expected

```
int eq( int a, int b )
{
    int a_eq_b = ( a == b );
    return a_eq_b;
}

bool eq( int a, int b )
{
    bool a_eq_b = ( a == b );
    return a_eq_b;
}
```

6.3. void* Type

- C allows automatic type conversion of void* to any pointer type
- C++ requires explicit type casting of void*

```
int main( void )
{
    int* x = malloc( 4 * sizeof(int) );
    free(x);
    return 0;
}

int main( void )
{
    int* x = (int*) malloc( 4 * sizeof(int) );
    free(x);
    return 0;
}
```
6. C++ Types

6.4. nullptr Literal

- C used the constant NULL to indicate a null pointer
- Part of the C standard library, not part of the language
- C++ includes a new nullptr literal for pointers

6.5. auto Type Inference

- C requires explicitly specifying the type in every variable declaration
- C++ includes the auto keyword for automatic type inference

```cpp
int avg( int x, int y )
{
int sum = x + y;
return sum / 2;
}

int main()
{
auto a = 1.0 + 2.5;  // type of a is double
auto b = 1.0f + 2.5f;  // type of b is float
auto c = avg( 10, 20 );  // type of c is int
return 0;
}
```
7. C++ Range-Based For Loop

- Iterating over arrays is very common and error prone
- C++ includes range-based for loops to simplify this common pattern
- Only works if compiler can figure out the size of the array

```cpp
int a[] = { 10, 20, 30, 40 };
int sum = 0;
for ( int v : a ) // v is a new temporary
    sum += v; // for each iteration of loop
int avg = sum / 4;
```

- Syntactic sugar for ...

```cpp
int a[] = { 10, 20, 30, 40 };
int sum = 0;
for ( int _i = 0; _i < 4; i++ ) {
    int v = a[i];
    sum += v;
}
int avg = sum / 4;
```
8. C++ Dynamic Allocation

- C dynamic allocation was handled by `malloc/free`
- Part of the C standard library, not part of the language
- C++ includes two new operators as part of the language
- The `new` operator is used to dynamically allocate variables
- The `delete` operator is used to deallocate variables
- These operators are “type safe” and are critical for object oriented programming

```cpp
int* a_p = new int;
*a_p = 42;
delete a_p;

int* b_p = new int[4];
b_p[0] = 1;
b_p[1] = 2;
b_p[2] = 3;
b_p[3] = 4;
delete[] b_p;

// struct Complex
// {
//   double real;
//   double imag;
// }

Complex* complex_p = new Complex;
complex_p->real = 1.5;
complex_p->imag = 3.5;
delete complex_p;
```
8. C++ Dynamic Allocation

- Revisiting earlier example for a function that appends a dynamically allocated node to a chain of nodes

```c
#include <cstddef>

struct Node
{
  int value;
  Node* next_p;
};

Node* append( Node* node_p, int value )
{
  Node* new_node_p = new Node;  /* Node* new_node_p = malloc( sizeof(Node) );
  new_node_p->value = value;
  new_node_p->next_p = node_p;
  return new_node_p;
}

int main( void )
{
  Node* node_p = NULL;
  node_p = append( node_p, 3 );
  node_p = append( node_p, 4 );
  delete node_p->next_p;      /* free( node_p->next_p );
  delete node_p;              /* free( node_p );
  return 0;
}
```