ECE 2400 Computer Systems Programming
Fall 2017

Topic 12: Transition from C to C++

School of Electrical and Computer Engineering
Cornell University

revision: 2017-10-23-01-13

1. 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

```
1  // contents of foo.h
2  // this avg rounds down
3  int avg( int x, int y )
4  {
5    int sum = x + y;
6    return sum / 2;
7  }
```

```
1  // contents of bar.h
2  // this avg rounds up
3  int avg( int x, int y )
4  {
5    int sum = x + y;
6    return (sum + 1) / 2;
7  }
```

```
1  #include "foo.h"
2  #include "bar.h"
3  #include <stdio.h>
4
5  int main( void )
6  {
7    printf("avg(2,3) = %d\n", avg(2,3) );
8    return 0;
9  }
```

http://cpp.sh/5vm7m

- 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;
}
```

http://cpp.sh/9wknn

- C++ namespaces extend the language to support named scopes
- Namespaces provide flexible ways to use specific scopes

```
// 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"
}
```

```
// 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"
}
```

```
#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;
}
```

http://cpp.sh/6dz5k

```
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;
}
```

- Namespaces are just syntactic sugar
- Useful way to group related struct and function definitions

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

```
namespace List {
  typedef struct
  {
    node_t* head_p;
  }
  list_t;
}
```

```
namespace List {
  void construct     ( list_t* list_p );
  void destruct      ( list_t* list_p );
  void push_front    ( list_t* list_p, int v );
  ...
}

int main( void )
{
  List::list_t list;
  List::construct( &list );
  List::push_front( &list, 12 );
  List::push_front( &list, 11 );
  List::push_front( &list, 10 );
  List::destruct( &list );
  return 0;
}
```

- 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

```
#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>`	`<cerrno>`	macro containing last error num
`<fenv.h>`	`<cfenv>`	floating-point access functions
`<float.h>`	`<cfloat>`	limits of float types
`<inttypes.h>`	`<cinttypes>`	formating 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

2. C++ Functions

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

```
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)

```
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

```
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

```
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

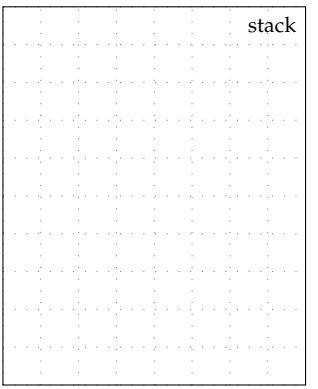
```
1  #include <cstdio>
2
3  enum round_mode_t
4  {
5    ROUND_MODE_FLOOR,
6    ROUND_MODE_CEIL,
7  };
8
9  int avg( int a, int b,
10           round_mode_t round_mode = ROUND_MODE_FLOOR )
11 {
12   int sum = a + b;
13   if ( round_mode == ROUND_MODE_CEIL )
14     sum += 1;
15   return sum / 2;
16 }
17
18 int main( void )
19 {
20   std::printf("avg( 5, 10 ) = %d\n", avg( 5, 10 ) );
21   return 0;
22 }                                          http://cpp.sh/3m4o
```

- 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

3. C++ References

- C provides pointers to indirectly refer to a variable

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



Aside:

```
// 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`

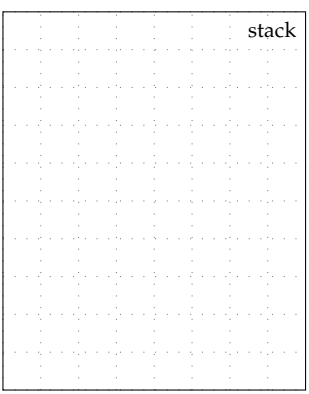
```
int&   a    // reference to a variable of type int
char&  b    // reference to a variable of type char
float& c    // reference to a variable of type float
```

- 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

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

- For the most part, references act like syntatic 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

```
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);
```

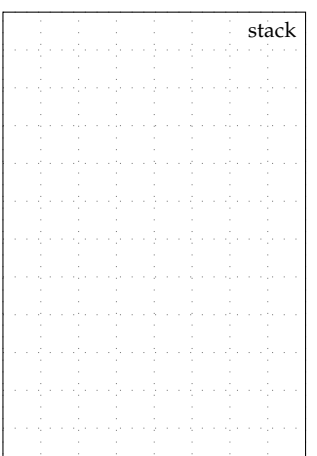


- Using pointers for call-by-reference

```
1  void sort2( int* x_ptr,
2              int* y_ptr )
3  {
4    if ( (*x_ptr) > (*y_ptr) ) {
5      int temp = *x_ptr;
6      *x_ptr   = *y_ptr;
7      *y_ptr   = temp;
8    }
9  }
```

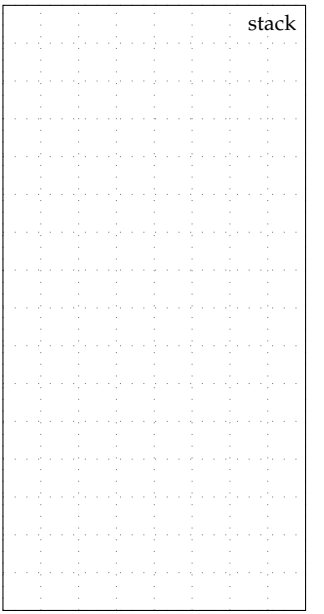
- Using references for call-by-reference

```
1  void sort2( int& x, int& y )
2  {
3    if ( x > y ) {
4      int temp = x;
5      x = y;
6      y = temp;
7    }
8  }
9
10 int main( void )
11 {
12   int a = 9;
13   int b = 5;
14   sort( a, b );
15   return 0;
16 }
```



http://cpp.sh/3f77d

Draw stack frame diagram corresponding to the execution of this program




```
1  void avg( int& result,
2            int x, int y )
3  {
4    int sum = x + y;
5    result = sum / 2;
6  }
7
8  int main( void )
9  {
10   int a = 10;
11   int b = 20;
12   int c;
13   avg( c, a, b );
14   return 0;
15 }
```

- 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

```
1  void blur( image_t* out, const image_t& in );
2
3  int main( void )
4  {
5    image_t in  = /* initialize */
6    image_t out = /* initialize */
7    blur( &out, in );
8    return 0;
9  }
```

4. C++ Exceptions

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

```
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

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

int main()
{
  int result = days_in_month( 7 );

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

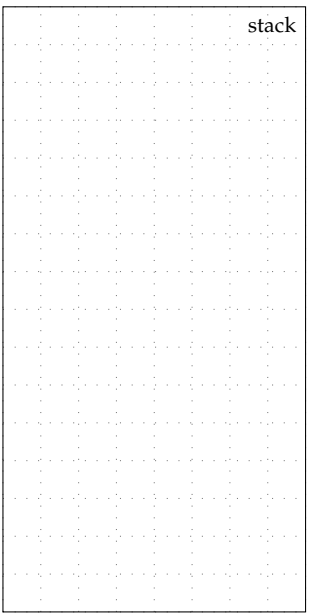
- C++ exceptions enable cleanly throwing and catching errors

```
#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 "Invalid month!";
  }
  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 ( const char* e ) {
    std::printf( "ERROR: %s\n", e );
    return -1;
  }

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



http://cpp.sh/9v3km

- Can throw variable of any type (e.g., integers, structs)
- Can catch and rethrow exceptions
- Uncaught exceptions will terminate the program

5. C++ Types

Small changes to three kinds of types

- `struct` types
- `bool` types
- `void*` types

5.1. `struct` Types

- C++ supports a simpler syntax for declaring `struct` types

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

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

```cpp
struct Complex
{
  double real;
  double imag;
};


int main( void )
{
  Complex 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

5.2. bool Types

- 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;
}
```

5.3. void* Types

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

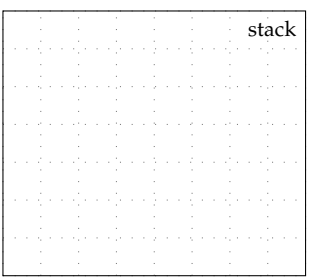
```
int main( void )
{
  int* x = malloc( 4 * sizeof(int) );
  free(x);
  return 0;
}
```

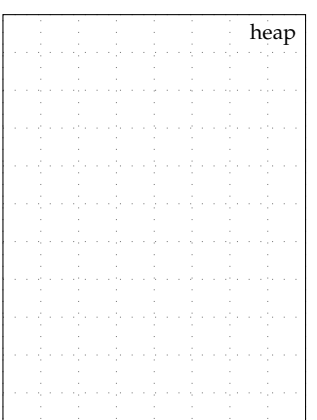
```
int main( void )
{
  int* x = (int*) malloc( 4 * sizeof(int) );
  free(x);
  return 0;
}
```

6. C++ 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

```
1  int* a_p = new int;
2  *a_p = 42;
3  delete a_p;
4
5  int* b_p = new int[4];
6  b_p[0] = 1;
7  b_p[1] = 2;
8  b_p[2] = 3;
9  b_p[3] = 4;
10 delete[] b_p;
11
12 // struct Complex
13 // {
14 //   double real;
15 //   double imag;
16 // };
17
18 Complex* complex_p = new Complex;
19 complex_p->real = 1.5;
20 complex_p->imag = 3.5;
21 delete complex_p;
```





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

```
#include <cstddef>

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

Node* append( Node* node_p, int value )
{
  Node* new_node_p =           // Node* new_node_p =
    new Node;                  //   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_pt );
  delete node_p;               // free( node_p );
  return 0;
}
```

- Revisiting earlier example for a function that dynamically allocates and then randomly initializes an array of integers

```
#include <cstddef>

int* rand_array( size_t size )
{
  int* x =                  // int* x =
    new int[size];          //   malloc( size * sizeof(int) );

  for ( size_t i=0; i<size; i++ )
    x[i] = rand() % 100;

  return x;
}

int main( void )
{
  int* a = rand_array(3);
  delete[] a;               // free(a);
  return 0;
}
```