

ECE 2400 Computer Systems Programming

Spring 2026

Topic 11: Transition to C++

School of Electrical and Computer Engineering
Cornell University

revision: 2026-03-16-12-16

1	Why C++?	3
1.1.	Procedural Programming	3
1.2.	T12: Object-Oriented Programming	4
1.3.	T13: Generic Programming	4
1.4.	T14: Concurrent Programming	5
1.5.	C vs. C++	6
2	C++ Namespaces	7
3	C++ Functions	12
4	C++ References	15
5	C++ Exceptions	20
6	C++ Types	23
6.1.	struct Types	23
6.2.	bool Type	24
6.3.	void* Type	24

6.4.	<code>nullptr</code> Literal	25
6.5.	<code>auto</code> Type Inference	25
7	C++ Range-Based For Loop	26
8	C++ Dynamic Allocation	27

Copyright © 2026 Anne Bracy. All rights reserved. This handout was prepared by Prof. Anne Bracy at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming (derived from previous handouts prepared and copyrighted by Prof. Christopher Batten). Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.

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

```
1  int avg( int x, int y )
2  {
3      int sum = x + y;
4      return sum / 2;
5  }
```

```
1  int main( void )
2  {
3      int c = avg(2,3);
4      return 0;
5  }
```

1.2. T12: 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

```
1  typedef struct
2  {
3      // implementation specific
4  }
5  slist_int_t;
6
7  void slist_int_construct ( slist_int_t* this );
8  void slist_int_destruct ( slist_int_t* this );
9  void slist_int_push_front ( slist_int_t* this, int v );
10 ...
```

1.3. T13: 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

```

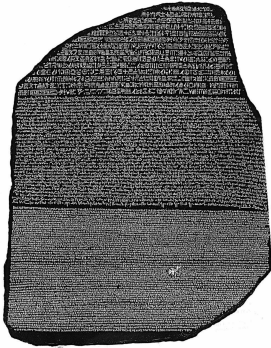
1  #define SPECIALIZE_SLIST_T( T )                               \
2                                                                    \
3  typedef struct                                               \
4  {                                                             \
5      /* implementation specific */                             \
6  }                                                               \
7  slist_ ##T## _t;                                             \
8                                                                    \
9  void slist_ ##T## _construct ( slist_ ##T## _t* this );     \
10 void slist_ ##T## _destruct ( slist_ ##T## _t* this );     \
11 void slist_ ##T## _push_front ( slist_ ##T## _t* this, T v ); \
12 ...                                                           \
13                                                                    \
14 SPECIALIZE_SLIST_T( int   )
15 SPECIALIZE_SLIST_T( float )

```

1.4. T14: 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

1.5. 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**

```
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 }
```

- Unclear which version of avg will be used
- Try it!causes compile-time “redefinition” error

- Traditional approach in C is to use prefixes
- Can create cumbersome syntactic overhead

```
1 // contents of foo.h           1 // contents of bar.h
2 // this avg rounds down       2 // this avg rounds up
3 int foo_avg( int x, int y )   3 int bar_avg( int x, int y )
4 {                               4 {
5     int sum = x + y;           5     int sum = x + y;
6     return sum / 2;            6     return (sum + 1) / 2;
7 }                               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 (rnd down)\n", foo_avg(2,3) );
8     printf("avg(2,3) = %d (rnd up)\n",   bar_avg(2,3) );
9     return 0;
10 }
```

(See zybook Lecture code in section 11.1)

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

```
1 // contents of foo.h           1 // contents of bar.h
2 namespace foo {                2 namespace bar {
3
4 // this avg rounds down        4 // this avg rounds up
5 int avg( int x, int y )        5 int avg( int x, int y )
6 {                               6 {
7     int sum = x + y;           7     int sum = x + y;
8     return sum / 2;           8     return (sum + 1) / 2;
9 }                               9 }
10
11 // Other code in              11 // Other code in
12 // namespace uses "avg"       12 // namespace uses "avg"
13 }                               13 }
```

```
1 #include "foo.h"
2 #include "bar.h"
3 #include <stdio.h>
4
5 int main( void )
6 {
7     printf("avg(2,3) = %d (rnd down)\n", foo::avg(2,3) );
8     printf("avg(2,3) = %d (rnd up)\n",   bar::avg(2,3) );
9     return 0;
10 }                                     (zyBook "Lab" 11.1.1)
```

```
1 int main( void )
2 {
3     using namespace foo;
4     printf("avg(2,3) = %d (rnd down)\n", avg(2,3) );
5     printf("avg(2,3) = %d (rnd up)\n",   bar::avg(2,3) );
6     return 0;
7 }                                     (zyBook "Lab" 11.1.2)
```

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

```
1 namespace ListInt {           1 namespace ListInt {
2     typedef struct _node_t    2     typedef struct
3     {                          3     {
4         int                    4         node_t* head_p;
5         struct _node_t* next_p; 5     }
6     }                          6     list_t;
7     node_t;                    7 }
8 }
```

```
1 namespace ListInt {
2     void construct ( list_t* this );
3     void destruct  ( list_t* this );
4     void push_front( list_t* this, int v );
5     ...
6 }
7
8 int main( void )
9 {
10    ListInt::list_t list;
11    ListInt::construct ( &list );
12    ListInt::push_front( &list, 12 );
13    ListInt::push_front( &list, 11 );
14    ListInt::push_front( &list, 10 );
15    ListInt::destruct  ( &list );
16    return 0;
17 }
```

- 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

```
1 #include "foo.h"
2 #include "bar.h"
3 #include <cstdio>
4
5 int main( void )
6 {
7     std::printf("avg(2,3) = %d (rnd down)\n", foo::avg(2,3) );
8     std::printf("avg(2,3) = %d (rnd up)\n",   bar::avg(2,3) );
9     return 0;
10 }
```

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

3. C++ Functions

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

```
1 int avg ( int x, int y );
2 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)

```
1 int avg( int x, int y )
2 {
3     int sum = x + y;
4     return sum / 2;
5 }
6
7 int avg( int x, int y, int z )
8 {
9     int sum = x + y + z;
10    return sum / 3;
11 }
12
13 int main( void )
14 {
15     // Will call definition of avg with 2 parameters
16     int a = avg( 10, 20 );
17
18     // Will call definition of avg with 3 parameters
19     int b = avg( 10, 20, 25 );
20
21     return 0;
22 }
```

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

```
1 int avg ( int x, int y );
2 double favg( double x, double y );
```

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

```
1 int avg( int x, int y )
2 {
3     int sum = x + y;
4     return sum / 2;
5 }
6
7 double avg( double x, double y )
8 {
9     double sum = x + y;
10    return sum / 2;
11 }
12
13 int main( void )
14 {
15     // Will call definition of avg with int parameters
16     int a = avg( 10, 20 );
17
18     // Will call definition of avg with double parameters
19     double b = avg( 7.5, 20 );
20
21     return 0;
22 }
```

- **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_FLOOR,
6      ROUND_CEIL,
7  };
8
9  int avg( int a, int b,
10         round_mode_t round_mode = ROUND_FLOOR )
11  {
12      int sum = a + b;
13      if ( round_mode == ROUND_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      std::printf("avg( 5, 10 ) = %d\n", avg( 5, 10, ROUND_CEIL ) );
22      return 0;
23  }
```

(See zybook Code Example 11.1.2)

- 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

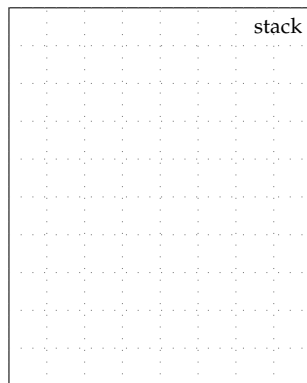
4. C++ References

- C provides pointers to indirectly refer to a variable

```

□□□ 01 int a = 3;
□□□ 02 int* b = &a;
□□□ 03 *b = 5;
□□□ 04 int c = *b;
□□□ 05 int* d = &(*b);

```



- Pointer syntax can sometimes be cumbersome
- C++ **references** are mostly syntactic sugar for pointers although with some important semantic restrictions
- 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

```

1 int& a // reference to a variable of type int
2 char& b // reference to a variable of type char
3 float& c // reference to a variable of type float

```

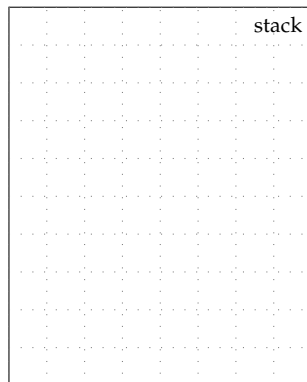
- Be careful – three very different uses of the & symbol!
 - Boolean AND operator `int d = b && c;`
 - Address-of operator `int* e = &a;`
 - Reference type `int& f`

(and there are actually even more!)

Four rules of references

1. References *must* be initialized to a valid variable
2. Once initialized, a reference cannot be changed to refer to a different variable
3. Always an **implicit address-of operator** when initializing a reference
4. Always an **implicit dereference operator** when using a reference

```
□□□ 01 int a = 3; // int a = 3;
□□□ 02 int& b = a; // int* b = &a;
□□□ 03 b = 5; // *b = 5;
□□□ 04 int c = b; // int c = *b;
□□□ 05 int* d = &b; // int* d = &(*b);
```



These rules lead to the following corollaries:

- References cannot use a variable declaration statement
- References cannot be uninitialized or NULL
- No pointers to references, arrays of references, references to references

```
1 int& a; // illegal (cannot use variable decl stmt)
2 int& b = NULL; // illegal (cannot be NULL)
3 int&* c; // illegal (no pointers to references)
4 int& d[10]; // illegal (no arrays of references)
```

Technically, we are focusing on **lvalue** references, **rvalue** references are a more advanced feature of C++

- **Call-by-pointer** uses pointers as function parameters

```

1 void swap( int* x_ptr, int* y_ptr )
2 {
3     int temp = *x_ptr;
4     *x_ptr   = *y_ptr;
5     *y_ptr   = temp;
6 }
7
8 int main( void )
9 {
10    int a = 9;
11    int b = 5;
12    swap( &a, &b );
13    return 0;
14 }

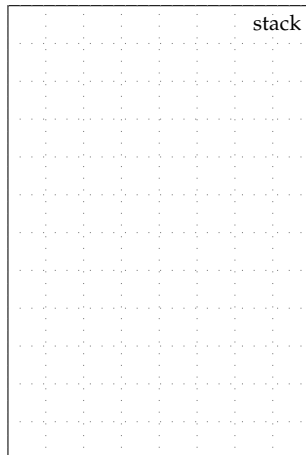
```

- **Call-by-reference** uses references as function parameters

```

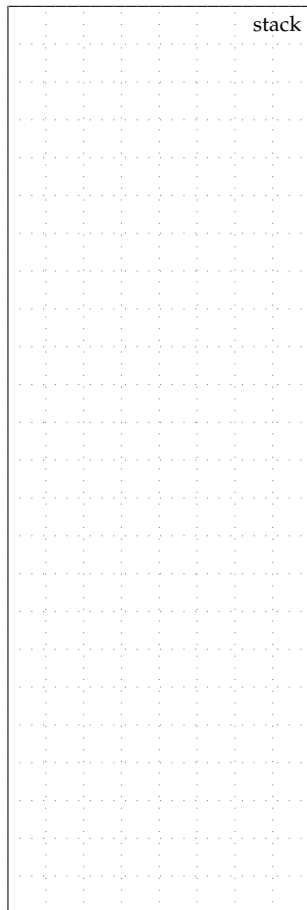
□□□ 01 void swap( int& x, int& y )
□□□ 02 {
□□□ 03     int temp = x;
□□□ 04     x = y;
□□□ 05     y = temp;
□□□ 06 }
□□□ 07
□□□ 08 int main( void )
□□□ 09 {
□□□ 10     int a = 9;
□□□ 11     int b = 5;
□□□ 12     swap( a, b );
□□□ 13     return 0;
□□□ 14 }

```



Draw a state diagram corresponding to the execution of this program

```
□□□ 01 typedef struct
□□□ 02 {
□□□ 03     double re;
□□□ 04     double im;
□□□ 05 }
□□□ 06 complex_t;
□□□ 07
□□□ 08 complex_t cmul( complex_t& x,
□□□ 09                 complex_t& y )
□□□ 10 {
□□□ 11     complex_t z;
□□□ 12     z.re = (x.re*y.re) - (x.im*y.im);
□□□ 13     z.im = (x.re*y.im) + (x.im*y.re);
□□□ 14     return z;
□□□ 15 }
□□□ 16
□□□ 17 int main( void )
□□□ 18 {
□□□ 19     complex_t a;
□□□ 20     a.re = 2.0;
□□□ 21     a.im = 0.5;
□□□ 22
□□□ 23     complex_t b;
□□□ 24     b.re = 10.0;
□□□ 25     b.im = 8.0;
□□□ 26
□□□ 27     complex_t c;
□□□ 28     c = cmul( a, b );
□□□ 29     return 0;
□□□ 30 }
```



- **Call-by-const-reference** uses constant references as function parameters to ensure the callee cannot change the value stored on the caller's stack frame

```
1 complex_t cmul( const complex_t& x, const complex_t& y )
2 {
3     complex_t z;
4     z.re = ( x.re * y.re ) - ( x.im * y.im );
5     z.im = ( x.re * y.im ) + ( x.im * y.re );
6     return z;
7 }
```

Three key uses of references

1. **Call-by-reference** to enable callee to modify variables on the caller's stack frame with less syntactic overhead compared to call-by-pointer (*rare, should avoid unless function name makes it very obvious*)
 2. **Call-by-const-reference** to combine the syntax and safety of call-by-value with the efficiency of call-by-pointer (*very common*)
 3. **Operator overloading** to enable applying built-in operators to user-defined types with elegant syntax (*very common*)
- **Do not overuse references!**
 - Avoid arbitrarily using references instead of pointers
 - Only use references for a very specific purpose, usually for call-by-const-reference or operator overloading

5. C++ Exceptions

- When handling errors in C, we can return an **error value**

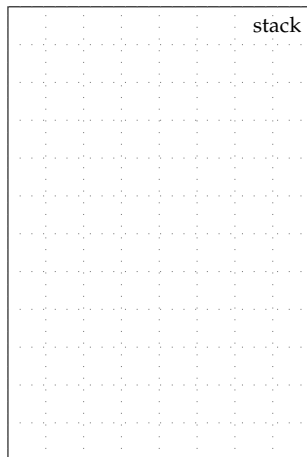
```
1 int days_in_month( int month )
2 {
3     if( (month < 1) || (month > 12) )
4         return -1;
5
6     switch ( month )
7     {
8         case 1: return 31;
9         case 1: return 28;
10        ...
11        case 12: return 31;
12    }
13 }
```

- When handling errors in C, we can **assert terminating the program**

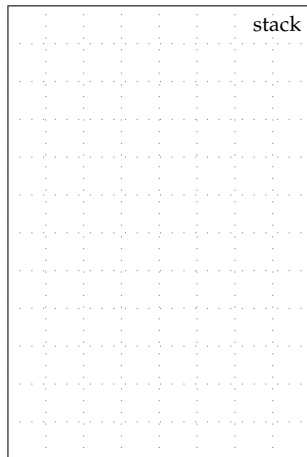
```
1 int days_in_month( int month )
2 {
3     assert( (month >= 1) && (month <= 12) );
4
5     switch ( month )
6     {
7         case 1: return 31;
8         case 1: return 28;
9         ...
10        case 12: return 31;
11    }
12 }
```

- C++ **exceptions** enable global non-linear control flow

```
□□□ 01 int x = 1;
□□□ 02
□□□ 03 try {
□□□ 04     int y = 10;
□□□ 05     if ( x )
□□□ 06         throw -1;
□□□ 07     y = 11;
□□□ 08     x = 2;
□□□ 09 }
□□□ 10 catch ( int e ) {
□□□ 11     x = e;
□□□ 12 }
```

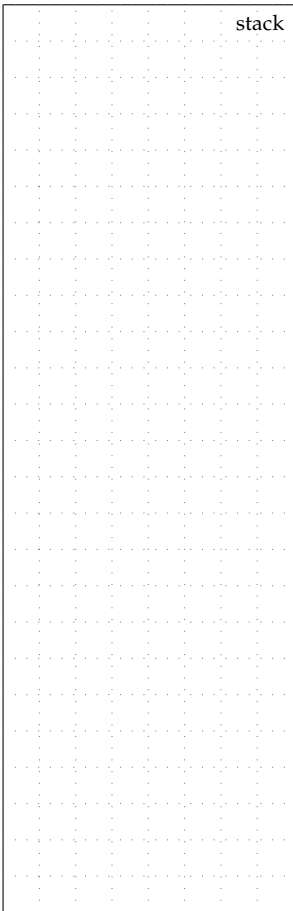


```
□□□ 01 int x = 0;
□□□ 02
□□□ 03 try {
□□□ 04     int y = 10;
□□□ 05     if ( x )
□□□ 06         throw -1;
□□□ 07     y = 11;
□□□ 08     x = 2;
□□□ 09 }
□□□ 10 catch ( int e ) {
□□□ 11     x = e;
□□□ 12 }
```



- C++ **exceptions** enable cleanly throwing and catching errors

```
01 int days_in_month( int month )
02 {
03     if ((month < 1) || (month > 12 ))
04         throw -1;
05
06     switch ( month )
07     {
08         case 1: return 31;
09         case 2: return 28;
10         ...
11         case 12: return 31;
12     }
13 }
14
15 void bar()
16 {
17     int days = days_in_month( 13 );
18 }
19
20 void foo()
21 {
22     bar();
23 }
24
25 int main( void )
26 {
27     try {
28         foo();
29     }
30     catch ( int e ) {
31         // display error message
32         return e;
33     }
34     return 0;
35 }
```



(See zybook Code Example 11.1.3)

- 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 type, 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

```
1  typedef struct                1  struct Complex
2  {                               2  {
3      double real;                3      double real;
4      double imag;                4      double imag;
5  }                               5  };
6  complex_t;                       6
7                                   7
8  int main( void )                8  int main( void )
9  {                               9  {
10     complex_t complex;          10     Complex complex;
11     complex.real = 1.5;          11     complex.real = 1.5;
12     complex.imag = 3.5;          12     complex.imag = 3.5;
13     return 0;                    13     return 0;
14 }                               14 }
```

- 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

```
1 int eq( int a, int b )           1 bool eq( int a, int b )
2 {                               2 {
3     int a_eq_b = ( a == b );    3     bool a_eq_b = ( a == b );
4     return a_eq_b;             4     return a_eq_b;
5 }                               5 }
```

6.3. void* Type

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

```
1 int main( void )
2 {
3     int* x = malloc( 4 * sizeof(int) );
4     free(x);
5     return 0;
6 }

1 int main( void )
2 {
3     int* x = (int*) malloc( 4 * sizeof(int) );
4     free(x);
5     return 0;
6 }
```

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

```
1 int lt( int x, int y ) { return x < y; }
2
3 int main( void )
4 {
5     auto cmp1 = &lt; ; // with auto
6     int (*cmp2)( int, int ) = &lt; ; // without auto
7
8     int result1 = cmp( 3, 4 );
9     int result2 = cmp( 5, 7 );
10    return 0;
11 }
```

- **Do not overuse auto!**
- Avoid arbitrarily using auto instead of explicitly specifying types
- Explicitly specifying types is an important form of documentation
- Do not use auto in a function parameter list, struct field types
- Only use auto for a very specific purpose, usually for very complicated types

```
1 auto a = 1 + 2; // not acceptable
2 auto b = "foo"; // not acceptable
3 auto c = avg( 10, 20 ); // not acceptable
```

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

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

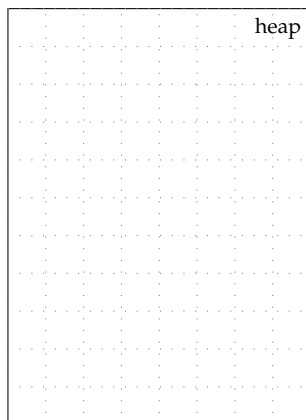
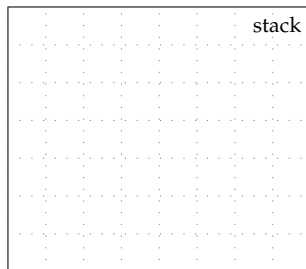
- Syntactic sugar for ...

```
1 int a[] = { 10, 20, 30, 40 };
2 int sum = 0;
3 for ( int i = 0; i < 4; i++ ) {
4     int v = a[i];
5     sum += v;
6 }
7 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

```
□□□ 01 int* a_ptr = new int;  
□□□ 02 *a_ptr = 42;  
□□□ 03 delete a_ptr;  
□□□ 04  
□□□ 05 int* b_ptr = new int[4];  
□□□ 06 b_ptr[0] = 1;  
□□□ 07 b_ptr[1] = 2;  
□□□ 08 b_ptr[2] = 3;  
□□□ 09 b_ptr[3] = 4;  
□□□ 10 delete[] b_ptr;  
□□□ 11  
□□□ 12 Complex* c_ptr = new Complex;  
□□□ 13 c_ptr->real = 1.5;  
□□□ 14 c_ptr->imag = 3.5;  
□□□ 15 delete c_ptr;
```



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

```
1  struct Node
2  {
3      int    value;
4      Node* next_p;
5  };
6
7  Node* prepend( Node* node_p, int v )
8  {
9      Node* new_node_p =           // Node* new_node_p =
10     new Node;                    // malloc( sizeof(Node) );
11     new_node_p->value = v;
12     new_node_p->next_p = node_p;
13     return new_node_p;
14 }
15
16 int main( void )
17 {
18     Node* node_p = nullptr;
19     node_p = prepend( node_p, 3 );
20     node_p = prepend( node_p, 4 );
21     delete node_p->next_p;        // free( node_p->next_pt );
22     delete node_p;                // free( node_p );
23     return 0;
24 }
```

- **Do not overuse new!**
- Avoid always using new with every struct (i.e., as in Java)
- If you *can* allocate a variable on the stack, you *should* allocate a variable on the stack
- Only use new for a very specific purpose, usually when you need more flexibility in the variable's lifetime or size