ECE 2400 Computer Systems Programming
Spring 2025

Topic 11: Transition to C++

School of Electrical and Computer Engineering
Cornell University

revision: 2025-03-17-10-27

zyBooks logo indicates readings and coding labs in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

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

```
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. 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

```
typedef struct
{
  // implementation specific
}
slist_int_t;

void slist_int_construct  ( slist_int_t* this );
void slist_int_destruct   ( slist_int_t* this );
void slist_int_push_front ( slist_int_t* this, int v );
...
```

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

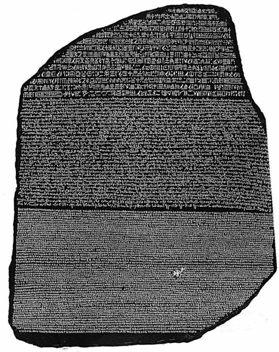
```
#define SPECIALIZE_SLIST_T( T )                                   \
                                                                  \
typedef struct                                                    \
{                                                                 \
  /* implementation specific */                                   \
}                                                                 \
slist_ ##T## _t;                                                  \
                                                                  \
void slist_ ##T## _construct  ( slist_ ##T## _t* this );          \
void slist_ ##T## _destruct   ( slist_ ##T## _t* this );          \
void slist_ ##T## _push_front ( slist_ ##T## _t* this, T v ); \
...

SPECIALIZE_SLIST_T( int   )
SPECIALIZE_SLIST_T( float )
```

1.4. T14: Concurrent Programming

- Programming is organized around computations that execute *concurrently* (i.e, computations exeute 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

```
// 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 will be used
- Try it! ....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;
}
```

(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

```
// 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;
}
```

(zyBook “Lab” 11.1.1)

```
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;
}
```

(zyBook “Lab” 11.1.2)

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

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

```
namespace ListInt {
  typedef struct
  {
    node_t* head_p;
  }
  list_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;
}
```

- 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

3. 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( void )
{
  // 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( void )
{
  // 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

```
#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 ) );
  std::printf("avg( 5, 10 ) = %d\n", avg( 5, 10, ROUND_MODE_C
  return 0;
}
```

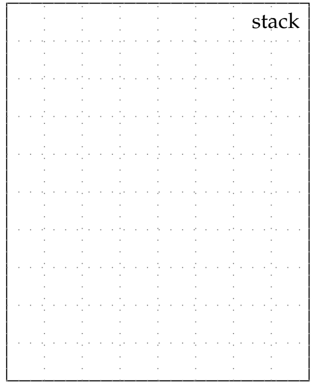
(See zybook Code Example 11.1.2)

- 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

```
int  a = 3;
int* b = &a;
*b = 5;
int  c = *b;
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`

```
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
```

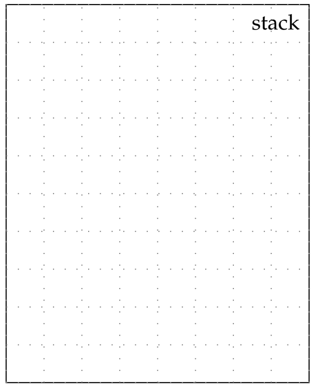
- Be careful – three very different uses of the `&` symbol!
  - Boolean AND operator `int  d = b && c;`
  - Address-of operator `int* e = &a;`
  - Reference type `int& f`

(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

```
int  a = 3;   // int  a = 3;
int& b = a;   // int* b = &a;
b = 5;        // *b = 5;
int  c = b;   // int  c = *b;
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

```
int&  a;        // illegal (cannot use variable decl stmt)
int&  b = NULL; // illegal (cannot be NULL)
int&* c;        // illegal (no pointers to references)
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

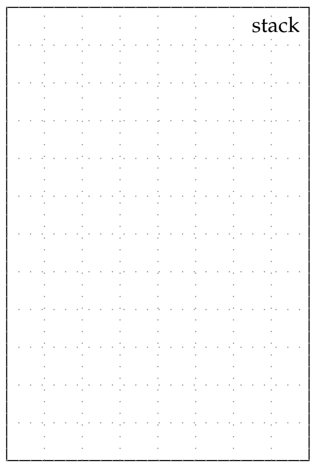
```cpp
void swap( int* x_ptr, int* y_ptr )
{
  int temp = *x_ptr;
  *x_ptr   = *y_ptr;
  *y_ptr   = temp;
}

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

- Call-by-reference uses references as function parameters

```cpp
void swap( int& x, int& y )
{
  int temp = x;
  x = y;
  y = temp;
}

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



Draw a state diagram corresponding to the execution of this program

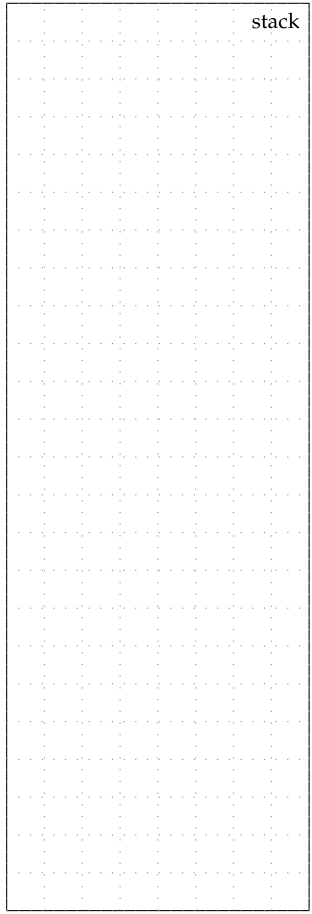
```cpp
typedef struct
{
  double re;
  double im;
}
complex_t;

complex_t cmul( complex_t& x,
                complex_t& y )
{
  complex_t z;
  z.re = (x.re*y.re) - (x.im*y.im);
  z.im = (x.re*y.im) + (x.im*y.re);
  return z;
}

int main( void )
{
  complex_t a;
  a.re =  2.0;
  a.im =  0.5;

  complex_t b;
  b.re = 10.0;
  b.im =  8.0;

  complex_t c;
  c = cmul( a, b );
  return 0;
}
```



- 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

```
complex_t cmul( const complex_t& x, const complex_t& y )
{
  complex_t z;
  z.re = ( x.re * y.re ) - ( x.im * y.im );
  z.im = ( x.re * y.im ) + ( x.im * y.re );
  return z;
}
```

Three key uses of references

1. **Call-by-reference** to enable callee to modify variables on the caller's stack frame with less syntatic 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

```
int days_in_month( int month )
{
  if( (month < 1) || (month > 12 ) )
    return -1;

  switch ( month )
  {
    case  1: return 31;
    case  1: return 28;
    ...
    case 12: return 31;
  }
}
```

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

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

  switch ( month )
  {
    case  1: return 31;
    case  1: return 28;
    ...
    case 12: return 31;
  }
}
```

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

```
int x = 1;

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

stack

```
int x = 0;

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

stack

- C++ exceptions enable cleanly throwing and catching errors

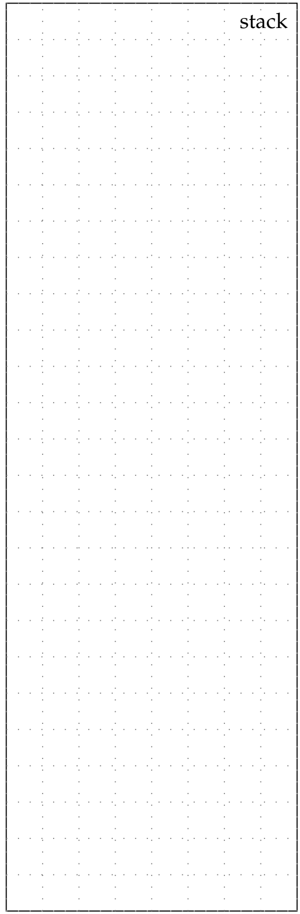
```
int days_in_month( int month )
{
  if ((month < 1) || (month > 12 ))
    throw -1;

  switch ( month )
  {
    case  1: return 31;
    case  2: return 28;
    ...
    case 12: return 31;
  }
}

void bar()
{
  int days = days_in_month( 13 );
}

void foo()
{
  bar();
}

int main( void )
{
  try {
    foo();
  }
  catch ( int e ) {
    // display error message
    return e;
  }
  return 0;
}
```



(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

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

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

```
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

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;
}
```

6.4. `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

```
int lt( int x, int y ) { return x < y; }

int main( void )
{
  // type of cmp is quite complex: int (*)( int, int )
  auto cmp = &lt;
  int result = cmp( 3, 4 );
  return 0;
}
```

- 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

```
auto a = 1 + 2;           // not acceptable
auto b = "foo";           // not acceptable
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

```
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;
```

- Syntatic sugar for ...

```
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;
```

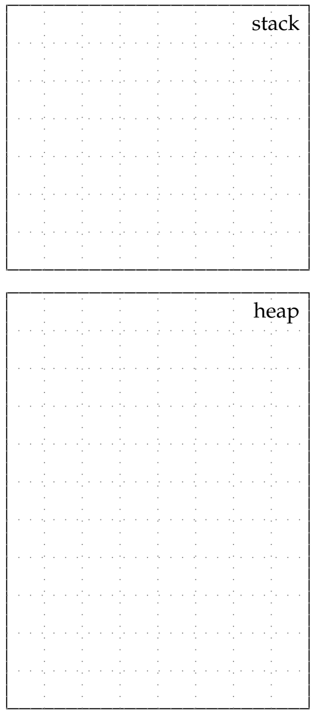
8. C++ Dynamic Allocation

- 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

```
int* a_ptr = new int;
*a_ptr = 42;
delete a_ptr;

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

Complex* c_ptr = new Complex;
c_ptr->real = 1.5;
c_ptr->imag = 3.5;
delete c_ptr;
```



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

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

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

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

- 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