CSc 28
Discrete Structures

Chapter 12
Function Parameters

Herbert G. Mayer, CSU CSC
Status 1/1/2021

Syllabus

- Summary
- Aliasing
- Parameter Overview
- Value Parameter
- Reference Parameter
- Value-Result Parameter
- Name Parameter
- In Out Parameter
- Operator Precedence

Overview

- Passing parameters is the key programming language tool for binding a caller's object to the callée
- The parameter on the calling side is the actual parameter (AP)
- In which case the actual may be an expression or a named object (variable), whose current value is passed
- The parameter on the called side (callée) is the formal parameter (FP); it is named, has a type, and the types of formal and actual must be compatible; depends on language
- The actual is bound to the formal at the place of call

Overview

- Binding lasts exactly for the duration of the call
- Association actual-to-formal is generally by position
- Association may also be by naming the formal at the place of call, e.g. in Ada, in which case order may be arbitrarily permuted, adding great possibilities for obfuscation ☺
- Types and numbers of actual and formal parameter (FP) generally must be compatible, subject to type compatibility- or type conversion rules of the language
- C allows a lesser number of actual parameters (AP) to be passed than formally declared (watch out!)

Overview

- Implementations of C therefore pass actuals in reverse textual order; see detail in Compiler class
- Theoretically, APs are passed on the run-time stack
- Physically, good optimizers pass APs in registers
- Practically, compiler passes the first few actual parameters in registers, remaining ones on the stack; note that the register resource is generally scarce!
- FPs are generally located on one side of the stack marker; locals are positioned on the opposite side of the stack marker
- Thus for addressing, one kind requiring negative, the other positive offsets from a base pointer register

Overview

```cpp
#include <iostream>

int add(int x, int y)
{
    return x + y;
}

int main()
{
    std::cout << add(4, 5) << std::endl;
    return 0;
}
```

x = 4

y = 5

Return value = 9

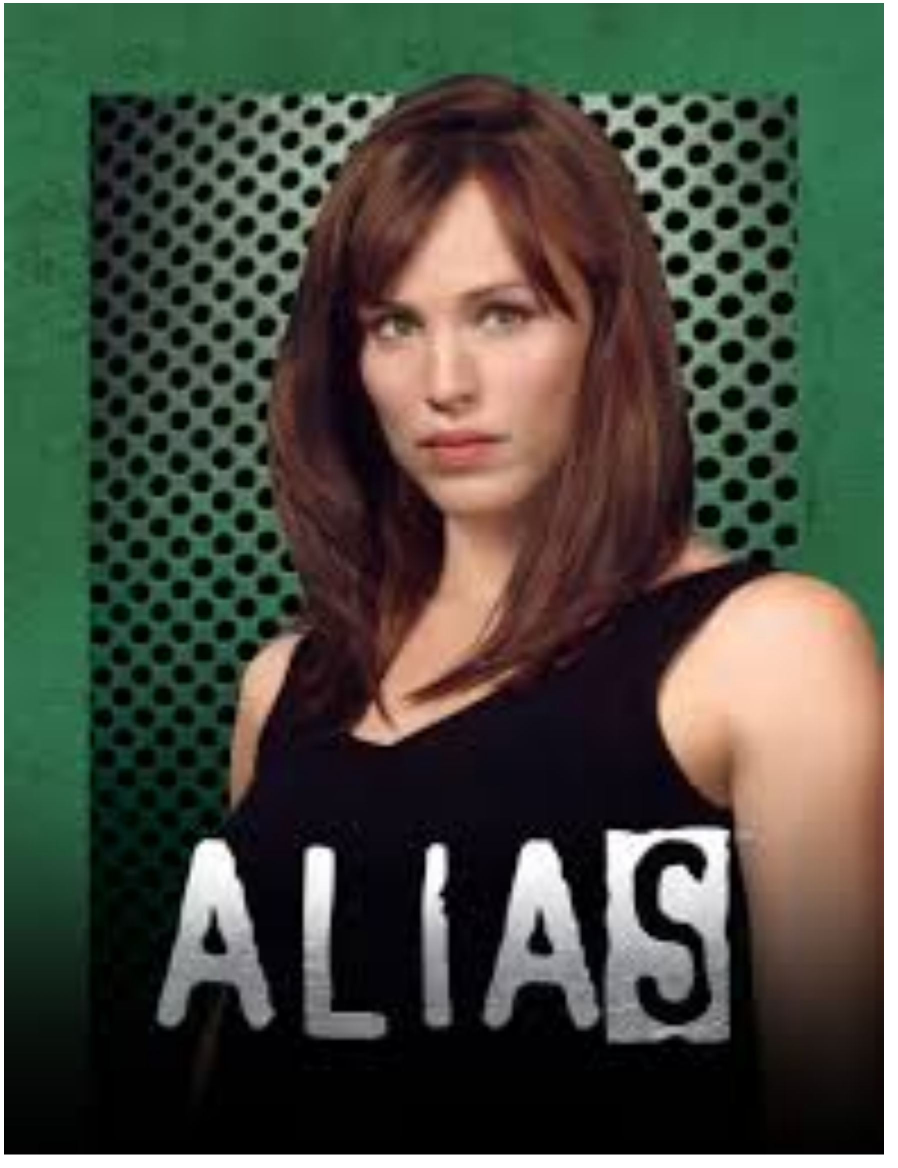
Overview

- Value Parameter (VP): Actual parameter (AP) provides initial value via a type-compatible expression. A modification to the bound formal parameter (FP) does not impact the AP
- Reference Parameter (RP): AP must be an addressable object! A modification to the bound FP immediately impacts the AP. Implemented by passing the address of the AP, which is a variable
- Value-Result Parameter (VRP): not needed for C++; AP must be an addressable object. A modification to the bound FP impacts the AP, but at the moment and place of return, not at the assignment! Implemented via copy-in copy-out!

Overview

- **Name parameter** (AKA call by name) in Algol60 is quite complex; not detailed here; way more complex than reference parameter passing
- **In parameter** parameter passing in Ada is similar to value parameters, but the formal may not be assigned, i.e. not changed!
- **Out parameter** parameter passing in Ada is similar to reference parameters, but no defined value is coming in at moment of call. Also, programmer may not rely on (expect) a specific implementation method
- **In out parameter** passing in Ada is similar to reference parameter passing

Aliasing



Aliasing

- Making one program object o1 accessible via a second name o2 is known as aliasing
- We say: object o2 is an alias of object o1, if both designate the same memory address
- The aliasing relation is reflexive: if o1 is and alias of o2, then o2 is also an alias of o1
- When o1 and o2 are aliases, and o1 is changed, then magically o2 also changes
- Surprises in SW are generally a sign of poor ☹ SWE
- Aliasing is not necessarily sign of poor programming!
- SW using aliasing is generally harder to comprehend, thus tends to be more error prone
- Aliasing is not a programming error; it is obfuscating practice; but often its use is a sin ☺

Aliasing

- Alias relations are typically established via reference parameters
- We won’t argue here whether pointers cause aliasing as well; they surely allow programmers to access objects differently than via their name proper!
- Example of aliasing in C++, assuming `ASSERT()`:

```
int glob_i      =      109;     // ooh global ☹ in C++
. . .
void foo1( int & loc_i )   // loc_i is a ref parameter
{ // foo1
    ASSERT( 109 == glob_i, "wrong value for glob_i" );
    ASSERT( 109 == loc_i,  "wrong parameter passing" );
    loc_i++;               // what will be changed?
    ASSERT( 110 == glob_i, "aliasing failed." );
} //end foo1
. . .
int main( void )
{ // main
    foo1( glob_i );        // the only call to foo1()
} //end main
```

Aliasing

- Pointers add further aliasing methods, even without parameterizing!
- When changing an object via pointers it is crucial to understand precisely, where that pointer points to!

```
int glob_i = 110;            // bad programmer ☺ A global!
. . .
void foo2( void )            // value parameter
{ // foo2
    int * p_i = & glob_i;  // set local pointer to glob_i

    ASSERT( 110 == glob_i, "error, glob_i should be 110" );
    ASSERT( 110 == *p_i,   "error, * to wrong 1. object" );
    (*p_i)++;               // *p_i is alias of glob_i
    ASSERT( 111 == glob_i, "error glob_i value" );
    ASSERT( 111 == *p_i,   "error, * to wrong 2. object" );
    ++(*p_i);               // another way to change
    ASSERT( 112 == glob_i, "error in pre-inc. via *" );
    *p_i++;                 // this fails the ASSERT
    ASSERT( 113 == glob_i, ”precedence + associativity!" );
} //end foo2
```

Parameter Overview

	value	ref	val-res	in	out	in out	name
AP may be expression	y	n	n	y	n	y	y
AP may init FP	y	y	y	n	n	y	y
AP must init FP	y	y	y	y	n	y	n
FP may be ref'ed	y	y	y	y	n	y	y
FP may be changed	y	y	y	n	y	y	y
AP changes with FP inside	n	y	n	n	dep	dep	y
AP changed after ret	n	y	y	n	y	y	y

Value Parameter (VP)

- Actual parameter (AP) bound to a formal value parameter (VP) must provide an initial value to the formal parameter (FP)
- Initial value of FP is value of actual at moment of call; that initial actual may be expression or addressable object
- During the call, assignments to the formal VP are allowed; exception: IN parameters in Ada
- Assignments to the formal VP have no impact on AP during or after the call
- Implementation: must make a copy of the initial value of the AP and pass copy in register, or on the stack
- At place of return, discard the copy, which immediately voids changes to the FP during the call

Value Parameter in C++

- Traditional C requires parameter passing by value
- Except for function parameters of type array[]

  Arrays in C are passed by reference; generally implemented by passing the address of the actual array; is address of element [0]

  Not to be confused with pointer types! More below!

  So far the same in C++

- Pointer type parameters in C++ are passed by value; of course the object pointed to may very well change during call, yet keeping the pointer itself invariant
- C++ structs are passed by value, requiring a copy of the actual struct; that may be a large amount of data space
- How can a programmer pass an array[] by value?
- *Pack* an array[] as a field into a struct: Consumes identical amount of space as original array[]
- Will all be copied wholesale at place and moment of call

Value Parameter in C++

- C++ inherits C's parameter passing methods
- Hence the C++ default parameter passing method is also by value
  - Recall in C only and all arrays are passed by reference in function calls!
- But C++ also adds the explicit reference parameter classification
- Using the & type qualifier for formal reference parameter class
- Consequently, actual parameter for formal reference parameter:
  - Must be an addressable objects! Not expressions!
  - Actuals passed may have new values after function return!
  - In line with strict definition of reference parameter

C++ Specifications

Value Parameter Set-Up

```
// Specifications used below:

// ASSERT() macro used in several samples!
// different from C++ default provided ASSERT()

#define ASSERT( condition, msg, value )                  \
   if( ! ( condition ) ) {                               \
     printf( "<> error: %s. Instead: %d\n", msg, value );\
   } //end if

typedef struct { int field; } struct_tp;    // 1 field?
typedef int * int_ptr_tp;

int global1 = 109;      // yes, globals! Dubious programming
int global2 = 2013;     // more globals ☹
```

C++ int Value Parameter

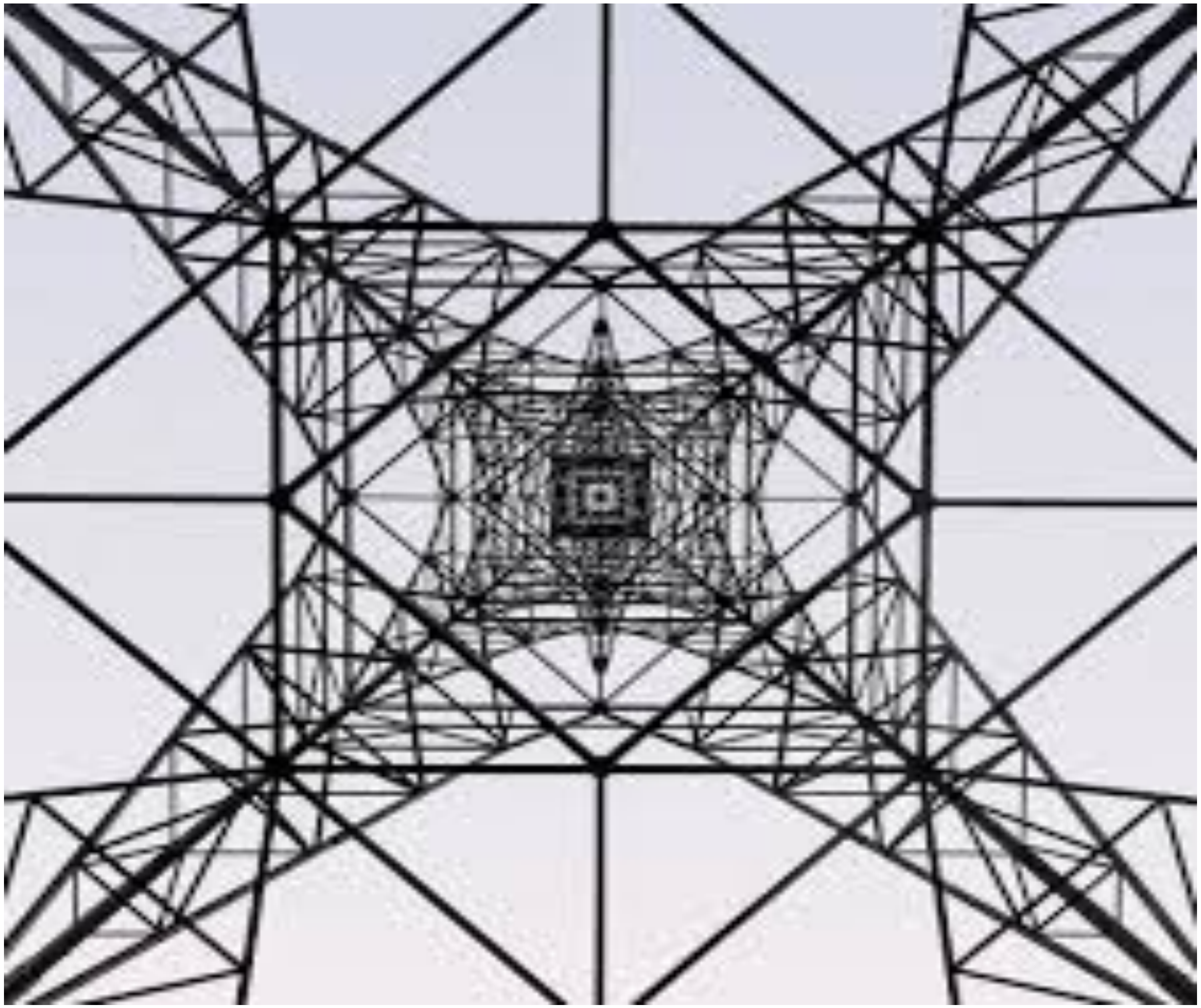
Value `int` Parameter in C++

```
// assignment to formal VP has no impact on AP
// must come in (be called) with: value = 109
void change_int_val( int value )
{ // change_int_val
    ASSERT( 109 == value,
      "in change_int_val() should be 109, is: ”, value );
    value++; // change formal parameter (FP)
    ASSERT( 110 == value,
      "in change_int_val() should be 110”, value );
} //end change_int_val

// call with value parameter
void int_val( void )    // <- start execution here!!
{ // int_val
    int local1 = 109;
    change_int_val( local1 );
    ASSERT( ( 109 == local1 ), “should be 109", local1 );
} //end int_val
```

C++ struct Value Parameter

Another Structure



Value `struct` Parameter in C++

```
// change formal value struct parameter
// should have no impact on actual parameter
void change_struct_val( struct_tp value )
{ // change_struct_val
    ASSERT( 109 == value.field,
       "in change_struct_val, expect 109", value.field );
    value.field++;   // change formal
    ASSERT( 110 == value.field,
       "in change_struct_val, expect 110", value.field );
} //end change_struct_val

void struct_val( void )    // <- start execution here!!
{ // struct_val
    struct_tp local1;      // see p. 17
    local1.field = 109;
    change_struct_val( local1 );
    ASSERT( ( 109 == local1.field ),
       "struct field should be 109", local1.field );
} //end struct_val
```

C++ Pointer Value Parameter

Value `pointer` Parameter in C++

```
// assignment to formal ptr type parameter,
// has has no impact on actual
void change_ptr_val( int_ptr_tp value )
{ // change_ptr_val
    ASSERT( 109 == *value,
       "in change_ptr_val should be 109", *value );
    value = & global2;  // formal parameter change!
    ASSERT( *value == 2013,
       "pointed-to val should be 2013", *value );
} //end change_ptr_val

void ptr_val( void )    // <- start execution here!!
{ // ptr_val
    int_ptr_tp local1 = & global1; // init 109; slide 17
    ASSERT( local1 == & global1,
       "before: should be & global1", & global1 );
    change_ptr_val( local1 );
    ASSERT( local1 == & global1,
       "Wrong pointer local1; points to:", *local1 );
    ASSERT( 109 == *local1,
       "pointed-to value should be 109", *local1 );
} //end ptr_val
```

Reference Parameters

Reference Parameter (RP)

- Actual reference parameter must be addressable object; it carries back new value after return
- I.e. the AP for RP passing must be a named (AKA addressable) object; it cannot be an expression! Not even a named object that is parenthesized
- AP does not have to be initialized (caution!), but in such cases the callée better abstain from referencing uninitialized objects before assignment!
- Goal is for callée to eventually compute and then return a new value in the AP; else RP use senseless
- Assignments to the formal RP have an immediate impact on the AP, since FP and AP are aliased to one another: they are and the same during call!

int Reference Parameters

Reference Parameter (RP)

- Aliasing can be source of hard to track errors!
- RP parameter passing enables aliasing!
- Implementation of RP done by passing the address of the AP
- Hence that AP has to be an addressable object
- Other implementations possible, specifically: passing address in register; must still provide an address of actual object!
- Effect of actual parameter passed in register will still adhere to reference parameter specification!

Reference Parameter in C++

```
#include <iostream>

#define ASSERT( condition, msg, value )                \
     if( ! ( condition ) ) {                           \
        printf( "<> error: %s. Instead: %d\n",        \
           msg, ( value ) );                           \
        error_count++;   //                            \
     } //end if

#define TRACK_NO_ERROR( msg )                          \
     if( ! error_count ) {                             \
         cout << "OK in: " << msg << endl;             \
     } //end if

#define MAX 10        // used for some array type defs
typedef struct {      // create struct with 1 int field
     int field;       // due to size = 8 bytes -> no padding
} struct_tp;

typedef int arr_tp[ MAX ];
typedef int * int_ptr_tp;
```

Reference Parameter in C++

```
int error_count;              // reset to 0 after each test
int global1;                  // will be set by init()
int global2;                  // also set by init()

struct_tp global_struct; // type struct_tp defined above

// init globally interesting things to defined values
// called by main(), so data ARE initialized
void init( void )
{ // init
    global1               = 109;
    error_count           = 0;
    global2               = 2013;
    global_struct.field = 109;
} //end init
```

Reference `int` Parameter in C++

```
// assignment to formal ref int impacts actual
void change_int_ref( int & value )
{ // change_int_ref
    ASSERT( value == 109, " ...", value );
    value++;                // change formal parameter
    ASSERT( 110 == global1, "...", global1 );
    ASSERT( 110 == value, "...", value );
} //end change_int_ref

void int_ref( void )
{ // int_ref                      // <- start execution here!!
    global1 = 109;
    change_int_ref( global1 );
    ASSERT( 110 == global1, "... NOT 110", global1 );
    TRACK_NO_ERROR( "int_ref()" );
} //end int_ref
```

... **Means**: there must be specific, meaningful message

Struct Reference Parameters

Reference `struct` Parameter in C++

```
// assign formal "ref struct parameter" impacts actual
void change_struct_ref( struct_tp & value ) // by ref!
{ // change_struct_ref
    ASSERT( value.field == 109, ". .", value.field );
    value.field++;                // change formal
    ASSERT( value.field == 110, ". .", value.field );
    ASSERT( 110 == global_struct.field, ". .",
      global_struct.field );  // aliasing!
    TRACK_NO_ERROR( "change_struct_ref()" );
} //end change_struct_ref

// assume global_struct.field initialized to 109; see p. 30
void struct_ref( void )        // <- start execution here!!
{ // struct_ref
    ASSERT( ( global_struct.field == 109 ), . . . ); // init
    change_struct_ref( global_struct );
    ASSERT( ( global_struct.field == 110 ),
      "struct field should be 110", global_struct.field );
} //end struct_ref
```

Pointer Reference Parameters

Reference `pointer` Parameter in C++

```
// assign to formal "ref ptr param" changes actual ptr
void change_ptr_ref( int_ptr_tp & value )
{ // change_ptr_ref
    ASSERT( 109 == *value,
       "pointed-to should still be 109", *value );
    value = & global2;   // formal ref parameter changed
    ASSERT( *value == 2013,
       "pointed-to should be 2013", *value );
} //end change_ptr_ref

void ptr_ref( void )                // <- start execution here!!
{ // ptr_ref                        // int_ptr_tp see p. 29
    int_ptr_tp local1 = & global1;   // global1 = 109
    ASSERT( 109 == *local1,
       "pointed-to should be 109 in ptr_ref", *local1 );
    change_ptr_ref( local1 );
    ASSERT( local1 == & global2, // local1 has changed!!
       "Wrong pointer; points to:", *local1 );
    ASSERT( 2013 == *local1,
       "pointed-to value should be 2013", *local1 );
} //end ptr_ref
```

Array Reference Parameters

Array Type Parameters in C++

```
// assign to formal "array param" impacts actual; no &
void change_arr_ref( arr_tp value ) // ref. is default!!
{ // change_arr_ref
    ASSERT( value[ 5 ] == 109,
       "value[ 5 ] should be 109", value[ 5 ] );
    value[ 5 ]++;  // formal value changed
    ASSERT( value[ 5 ] == 110,
       "value[5] should be 110 in change()”, value[5] );
} //end change_arr_ref

void arr_ref( void )    // <- start execution here!!
{ // arr_ref
    arr_tp value;   // is array of MAX=10 ints
    for( int i = 0; i < MAX; i++ ) { value[i] = 109; }
    ASSERT( 109 == value[ 5 ],
       "value[5] should be 109 in arr_ref", value[5] );
    change_arr_ref( value );
    ASSERT( 110 == value[ 5 ],
       "value[ 5 ] should be 110", value[ 5 ] );
} //end arr_ref
```

Value-Result Parameter in Fortran

- AP must be addressable object, to be bound to FP
- Assignments to FP are allowed in callée
- During the call, the AP is unchanged; note that this is quite different from RP passing!
- At the moment of return –and there may be many different places in the code– the last value assigned to the FP is copied back into the AP
- After the call, the AP is changed; note that this is quite different from VP passing!
- To be covered in Fortran programming class, or compiler class
- Won’t discuss here in detail

Name Parameter in Algol60

- Name parameter passing is the default, quirky parameter passing method in Algol60
- Hard to implement, but quite “powerful”
- The text of the AP substitutes the corresponding FP for any particular call; virtually substitutes!
- Can have strange side-effects
- A bit complicated to understand and implement, first done by Ingerman, using what he called: “thunks”
- Also not covered here in detail; belongs into more advanced Compiler class

Name Parameter in Algol60

```
Comment: Algol60 sample; definition of confuse()
procedure confuse( a, b )
   integer a, b
   begin
      b := 4;
      a := 5;
   end;

confuse( x[ i ], i ); comment: i = 33, x[33] = 10

Comment: as if confuse() had been coded like this:
procedure confuse( x[i], i )
   integer x[ i ], i
   begin
      i    := 4;
      x[4] := 5;
   end;

Comment: x[33] is unaffected, i = 4, and x[4] = 5
```

In Out Parameter in Ada

- Is general parameter passing method in Ada; similar to, but not identical to value-result parameter passing
- Is a combination of Ada’s in parameter and out parameter passing methods
- Careful with “similar to value-result parameter” passing: In Ada it is deliberatively left unspecified by language fiat, when (at which moment) the final value of the formal is copied back into the actual
- Any Ada program relying on a particular method is by definition an *erroneous program*!
- Also not discussed further

Operators

C++ Operator Precedence

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	a++ a--	Suffix/postfix increment and decrement	
	type() *type*{}	Functional cast	
	a()	Function call	
	a[]	Subscript	
	. ->	Member access	
3	++a --a	Prefix increment and decrement	Right-to-left
	+a -a	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	C-style cast	
	*a	Indirection (dereference)	
	&a	Address-of	
	sizeof	Size-of[note 1]	
	new new[]	Dynamic memory allocation	
	delete delete[]	Dynamic memory deallocation	
4	.* ->*	Pointer-to-member	Left-to-right
5	a*b a/b a%b	Multiplication, division, and remainder	
6	a+b a-b	Addition and subtraction	
7	<< >>	Bitwise left shift and right shift	
8	<=>	Three-way comparison operator (since C++20)	
9	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
10	== !=	For relational operators = and ≠ respectively	
11	&	Bitwise AND	
12	^	Bitwise XOR (exclusive or)	
13	|	Bitwise OR (inclusive or)	
14	&&	Logical AND	
15	||	Logical OR	
16	a?b:c	Ternary conditional[note 2]	Right-to-left
	throw	throw operator	
	=	Direct assignment (provided by default for C++ classes)	
	+= -=	Compound assignment by sum and difference	
	*= /= %=	Compound assignment by product, quotient, and remainder	
	<<= >>=	Compound assignment by bitwise left shift and right shift	
	&= ^= |=	Compound assignment by bitwise AND, XOR, and OR	
17	,	Comma	Left-to-right

C++ Operator Precedence

```
int num     = 1;
int arr[]   = { 33, 55, 77 };
int * pi    = arr;                // arr name = address of 1st

#define TRACK( msg, val ) printf( "num %d '%s' %6d, pi= %d, \
   arr[0] = %d, arr[1] = %d, arr[2] = %d\n",                \
   num++, msg, val, pi, arr[0], arr[1], arr[2] )
// note: no semicolon at end of macro body!!

int main( void )
{ // main
   -- A(x) below means: address of x
   TRACK( "    *pi", *pi );        // deref ^ to arr[0]
   TRACK( "(*pi)++", (*pi)++ ); // post-inc arr[0], NOT the ptr pi
   TRACK( "++(*pi)", ++(*pi) ); // pre-inc arr[0], NOT the ptr pi
   TRACK( "  *pi++",   *pi++ ); // get arr[0] post++ pi to A(arr[1])
   TRACK( "++*pi  ", ++*pi );   // pre-inc a[1]
   TRACK( "*++pi  ", *++pi );   // pre-inc ptr to a[2], fetch a[2]
   TRACK( "   pi++", pi++ );    // print addr, then inc by 4
   TRACK( "   ++pi", ++pi );    // inc word addr to after arr[2]

   return 0;
} //end main
```

C++ Operator Precedence

```
num 1 '    *pi'     33, pi= 135336, arr[0] = 33, arr[1] = 55, arr[2] = 77
num 2 '(*pi)++'     33, pi= 135336, arr[0] = 34, arr[1] = 55, arr[2] = 77
num 3 '++(*pi)'     35, pi= 135336, arr[0] = 35, arr[1] = 55, arr[2] = 77
num 4 '  *pi++'     35, pi= 135340, arr[0] = 35, arr[1] = 55, arr[2] = 77
num 5 '++*pi  '     56, pi= 135340, arr[0] = 35, arr[1] = 56, arr[2] = 77
num 6 '*++pi  '     77, pi= 135344, arr[0] = 35, arr[1] = 56, arr[2] = 77
num 7 '   pi++' 135344, pi= 135348, arr[0] = 35, arr[1] = 56, arr[2] = 77
num 8 '   ++pi' 135352, pi= 135352, arr[0] = 35, arr[1] = 56, arr[2] = 77
```

C++ Operators

- **Function call** constitutes just one of many operators in C++ with defined precedence (AKA **priority**)
- Not discussed in detail here, yet informative and impressive to just show total list, in priority order, for C++
- Below also for Java
- Also not to miss opportunity of good advice: "**When in doubt, parenthesize!**"
- Who would want to learn by heart over a dozen different precedence levels? ☺

Java Operator Precedence

Java Operator Precedence Table

<table>
<tr><th>Precedence</th><th>Operator</th><th>Type</th><th>Associativity</th></tr>
<tr><td>15</td><td>()
[]
·</td><td>Parentheses
Array subscript
Member selection</td><td>Left to Right</td></tr>
<tr><td>14</td><td>++
--</td><td>Unary post-increment
Unary post-decrement</td><td>Right to left</td></tr>
<tr><td>13</td><td>++
--
+
-
!
~
(*type*)</td><td>Unary pre-increment
Unary pre-decrement
Unary plus
Unary minus
Unary logical negation
Unary bitwise complement
Unary type cast</td><td>Right to left</td></tr>
<tr><td>12</td><td>*
/
%</td><td>Multiplication
Division
Modulus</td><td>Left to right</td></tr>
<tr><td>11</td><td>+
-</td><td>Addition
Subtraction</td><td>Left to right</td></tr>
<tr><td>10</td><td><<
>>
>>></td><td>Bitwise left shift
Bitwise right shift with sign extension
Bitwise right shift with zero extension</td><td>Left to right</td></tr>
<tr><td>9</td><td><
<=
>
>=
instanceof</td><td>Relational less than
Relational less than or equal
Relational greater than
Relational greater than or equal
Type comparison (objects only)</td><td>Left to right</td></tr>
<tr><td>8</td><td>==
!=</td><td>Relational is equal to
Relational is not equal to</td><td>Left to right</td></tr>
<tr><td>7</td><td>&</td><td>Bitwise AND</td><td>Left to right</td></tr>
<tr><td>6</td><td>^</td><td>Bitwise exclusive OR</td><td>Left to right</td></tr>
<tr><td>5</td><td>|</td><td>Bitwise inclusive OR</td><td>Left to right</td></tr>
<tr><td>4</td><td>&&</td><td>Logical AND</td><td>Left to right</td></tr>
<tr><td>3</td><td>||</td><td>Logical OR</td><td>Left to right</td></tr>
<tr><td>2</td><td>? :</td><td>Ternary conditional</td><td>Right to left</td></tr>
<tr><td>1</td><td>=
+=
-=
*=
/=
%=</td><td>Assignment
Addition assignment
Subtraction assignment
Multiplication assignment
Division assignment
Modulus assignment</td><td>Right to left</td></tr>
</table>

Larger number means higher precedence.

Java Operators

- **Java specifies 15 difference precedence levels for source operators**
- **Versus C++ 17 levels**
- **E.g. no more separate comma operator "," in Java, with exception for-loop initialization**
- ***Good to know* by heart, but *better* to "parenthesize when unsure"! Even when sure: For the reader!**

Summary

- C++ macro parameters are distinct from function parameters; by reference or by value does not apply!
- Reference Parameters may carry in an initial value to the callée; such a carry in value is provided by the actual object (not an expression) on the calling side
- Different from Value Parameters, Reference can carry out a new value from callée to calling function; value out will be that of the last assignment to formal parameter
- C only knows reference parameter passing for array type parameters
- C++ extends idea via explicit & operator ref to any type
- In Out parameter passing in Ada similar, but left unspecified, from when to when exactly the actual and formal become aliases

References

1. Revised Algol60 report: http://www.masswerk.at/algol60/report.htm
2. ADA reference manual: http://www.adaic.org/resources/add_content/standards/05rm/html/RM-TTL.html
3. Wiki reference parameter: https://en.wikipedia.org/wiki/Reference_(C%2B%2B)