



**CS-202**

# C++ Primer (continued)

**C. Papachristos**

Autonomous Robots Lab  
University of Nevada, Reno



# Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday
			Lab (8 Sections)	
	CLASS		CLASS	
PASS Session	PASS Session	<b>Project DEADLINE</b>	<b>NEW Project</b>	

Your 1<sup>st</sup> Lab is today Thursday 1/24.

Your 1<sup>st</sup> Project will be announced today Thursday 1/24.

- Project is graded.
- Project Deadline is next Wednesday night 1/30 @ 11:59 pm (*firm*).

# Today's Topics

Operators & Expressions

Statements & Flow Control

C++ Input / Output

Namespaces & Resolution

Scope & Resolution

Arrays

# Operators & Expressions

Precedence	Operator	Description	Associativity
1	::	<a href="#">Scope resolution</a>	Left-to-right
2	a++ a--	Suffix/postfix <a href="#">increment and decrement</a>	
	type() type{}	<a href="#">Functional cast</a>	
	a()	<a href="#">Function call</a>	
	a[]	<a href="#">Subscript</a>	
	. ->	<a href="#">Member access</a>	
3	++a --a	Prefix <a href="#">increment and decrement</a>	Right-to-left
	+a -a	Unary <a href="#">plus and minus</a>	
	! ~	<a href="#">Logical NOT</a> and <a href="#">bitwise NOT</a>	
	(type)	<a href="#">C-style cast</a>	
	*a	<a href="#">Indirection</a> (dereference)	
	&a	<a href="#">Address-of</a>	
	sizeof	<a href="#">Size-of</a> <sup>[note 1]</sup>	
	new new[]	<a href="#">Dynamic memory allocation</a>	
	delete delete[]	<a href="#">Dynamic memory deallocation</a>	
4	. * ->*	<a href="#">Pointer-to-member</a>	Left-to-right
5	a*b a/b a%b	<a href="#">Multiplication, division, and remainder</a>	
6	a+b a-b	<a href="#">Addition and subtraction</a>	
7	<< >>	Bitwise <a href="#">left shift and right shift</a>	
8	<=>	<a href="#">Three-way comparison operator</a> (since C++20)	
9	< <=	For <a href="#">relational operators</a> < and ≤ respectively	
	> >=	For <a href="#">relational operators</a> > and ≥ respectively	

Precedence	Operator	Description	Associativity
10	== !=	For <a href="#">relational operators</a> = and ≠ respectively	Left-to-right
11	&	<a href="#">Bitwise AND</a>	
12	^	<a href="#">Bitwise XOR</a> (exclusive or)	
13		<a href="#">Bitwise OR</a> (inclusive or)	
14	&&	<a href="#">Logical AND</a>	
15		<a href="#">Logical OR</a>	
16	a?b:c	<a href="#">Ternary conditional</a> <sup>[note 2]</sup>	Right-to-left
	throw	<a href="#">throw operator</a>	
	=	<a href="#">Direct assignment</a> (provided by default for C++ classes)	
	+= -=	<a href="#">Compound assignment</a> by sum and difference	
	*= /= %=	<a href="#">Compound assignment</a> by product, quotient, and remainder	
	<<= >>=	<a href="#">Compound assignment</a> by bitwise left shift and right shift	
	&= ^=  =	<a href="#">Compound assignment</a> by bitwise AND, XOR, and OR	
17	,	<a href="#">Comma</a>	Left-to-right

Source:

[https://en.cppreference.com/w/cpp/language/operator\\_precedence](https://en.cppreference.com/w/cpp/language/operator_precedence)

# Operators & Expressions

## Standard Arithmetic Operators

Left-to-Right Associativity, Standard rules of arithmetic Precedence

- Parentheses
- Multiplication (  $*$  ), Division (  $/$  ), Modulo (  $\%$  ) - Precedence Group 5
- Addition (  $+$  ), Subtraction (  $-$  ) - Precedence Group 6
- Exponents ... (Note: Do not use (  $^$  ) for exponents.)

# Operators & Expressions

## Standard Relational Operators

Testing for:

- Equality ( **==** ), Inequality ( **!=** )
- Less-Than ( **<** ), Higher-Than ( **>** )
- Less/Equal-To ( **<=** ), Higher/Equal-To ( **>=** )
- Evaluate to ( **true** ) or ( **false** )

# Operators & Expressions

## Standard Logical Operators

Evaluating:

- Logical AND ( **&&** ), OR ( **||** ), NOT( **!** )
- Evaluate to ( **true** ) or ( **false** )

# Operators & Expressions

## Standard Bitwise Operators

Useful to conduct Bitwise operations:

(Boolean , bit-by-bit operations on Registers)

- AND ( **&** ) , OR ( **|** ) , XOR ( **^** ) , NOT( **~** )
- Bitwise Shifting Left ( **<<** ) , Right ( **>>** )



# Operators & Expressions

## Operators (General)

A variety of operators in programming languages:

- Unary (1), Binary (2), Ternary (3)  
(depends on number of operands, i.e. things they operate on)

Represented by special symbolic characters

- ( **+** ) means **add** ( . , . ) , hence it is a Binary operator.

# Operators & Expressions

## Unary Operators

- Logical Negation ( **!** )  
( **! true** ) is **false**  
( **! false** ) is **true**
- Post-Increment ( **• ++** ) and Post-Decrement ( **• --** )  
( **x ++** ) evaluates to ( **x** ), **x** is increased by **1**  
( **x --** ) evaluates to ( **x** ), **x** is decreased by **1**
- Pre-Increment ( **++ •** ) and Pre-Decrement ( **-- •** )  
( **++ x** ) evaluates to ( **x + 1** ), **x** is increased by **1**  
( **-- x** ) evaluates to ( **x - 1** ), **x** is decreased by **1**

# Operators & Expressions

## Expressions

When simple units of *operands and operators are combined* into larger units, (always following the strict rules of precedence and associativity).

- Expression is each **aggregate computable unit** (simpler or larger).

## Conditional Ternary Operator ( ? )

Composed of Expressions:

```
(Test_Expression) ? (Evaluated_Expression_If_TRUE) : (Evaluated_Expression_If_FALSE)
```

```
5==7 ? printf("5 equals 7") : printf("5 does not equal 7");
```

```
int a = 10;
```

```
int b = (5==7) ? 1*a : -1*a ;
```

# Operators & Expressions

## Expressions

When simple units of *operands and operators are combined* into larger units, (always following the strict rules of precedence and associativity).

- Expression is each **aggregate computable unit** (simpler or larger).

## Conditional Ternary Operator ( ? )

Composed of Expressions:

```
(Test_Expression) ? (Evaluated_Expression_If_TRUE) : (Evaluated_Expression_If_FALSE)
```

```
5==7 ? printf("5 equals 7") : printf("5 does not equal 7");
```

```
int a = 10;
```

```
int b = (5==7) ? 1*a : -1*a ;
```

# Operators & Expressions

## Expressions

When simple units of *operands and operators are combined* into larger units, (always following the strict rules of precedence and associativity).

- Expression is each **aggregate computable unit** (simpler or larger).

## Conditional Ternary Operator ( ? )

Composed of Expressions:

`(Test_Expression) ? (Evaluated_Expression_If_TRUE) : (Evaluated_Expression_If_FALSE)`

`5==7 ? printf("5 equals 7") : printf("5 does not equal 7");`

`int a = 10;`

`int b = (5==7) ? 1*a : -1*a ;`

# Operators & Expressions

## Expressions

When simple units of *operands and operators are combined* into larger units, (always following the strict rules of precedence and associativity).

- Expression is each **aggregate computable unit** (simpler or larger).

## Conditional Ternary Operator ( ? )

Composed of Expressions:

(Test\_Expression) ? (Evaluated\_Expression\_If\_TRUE) : (Evaluated\_Expression\_If\_FALSE)

5==7 ? printf("5 equals 7") : printf("5 does not equal 7");

```
int a = 10;
```

```
int b = (5==7) ? 1*a : -1*a ;
```

# Operators & Expressions

## Expressions

When simple units of *operands and operators are combined* into larger units, (always following the strict rules of precedence and associativity).

➤ Expression is each **aggregate computable unit** (simpler or larger).

## Conditional Ternary Operator ( ? )

Composed of Expressions:

`(Test_Expression) ? (Evaluated_Expression_If_TRUE) : (Evaluated_Expression_If_FALSE)`

`5==7 ? printf("5 equals 7") : printf("5 does not equal 7");`

```
int a = 10;
```

```
int b = (5==7) ? 1*a : -1*a ;
```

A Complex Statement (Assignment followed by Ternary Expression)



# Operators & Expressions

## Expressions

When simple units of *operands and operators are combined* into larger units, (always following the strict rules of precedence and associativity).

- Expression is each **aggregate computable unit** (simpler or larger).

## Conditional Ternary Operator ( ? )

Composed of Expressions:

(Test\_Expression) ? (Evaluated\_Expression\_If\_TRUE) : (Evaluated\_Expression\_If\_FALSE)

```
5==7 ? printf("5 equals 7") : printf("5 does not equal 7");
```

```
int a = 10;  
int b [=] (5==7) ? 1*a : -1*a ;
```

Right-to-Left Associativity of Assignment operator



# Operators & Expressions

## Expressions

When simple units of *operands and operators are combined* into larger units, (always following the strict rules of precedence and associativity).

- Expression is each **aggregate computable unit** (simpler or larger).

## Conditional Ternary Operator ( ? )

Composed of Expressions:

`(Test_Expression) ? (Evaluated_Expression_If_TRUE) : (Evaluated_Expression_If_FALSE)`

`5==7 ? printf("5 equals 7") : printf("5 does not equal 7");`

`int a = 10;`

`int b = [(5==7) ? 1*a : -1*a];`

Ternary Expression: Evaluation of Test Expression

# Operators & Expressions

## Expressions

When simple units of *operands and operators are combined* into larger units, (always following the strict rules of precedence and associativity).

➤ Expression is each **aggregate computable unit** (simpler or larger).

## Conditional Ternary Operator ( ? )

Composed of Expressions:

`(Test_Expression) ? (Evaluated_Expression_If_TRUE) : (Evaluated_Expression_If_FALSE)`

`5==7 ? printf("5 equals 7") : printf("5 does not equal 7");`

`int a = 10;`

`int b = (5==7) ? 1*a : -1*a;`

Result: **false** → Evaluates to 2<sup>nd</sup> Expression

# Operators & Expressions

## Expressions

When simple units of *operands and operators are combined* into larger units, (always following the strict rules of precedence and associativity).

- Expression is each **aggregate computable unit** (simpler or larger).

## Conditional Ternary Operator ( ? )

Composed of Expressions:

`(Test_Expression) ? (Evaluated_Expression_If_TRUE) : (Evaluated_Expression_If_FALSE)`

`5==7 ? printf("5 equals 7") : printf("5 does not equal 7");`

`int a = 10;`

`int b = [-----  
          -10;-----]`

Evaluation of Ternary Expression is a double literal

# Operators & Expressions

## Operator Associativity

Kicks in when operators of the same precedence appear in an Expression.

Postfix operators: **++ --** (left to right)

Prefix operators: **++ --** (right to left)

Unary operators: **+ - !** (right to left)

**\* / %** (left to right)

**+ -** (left to right)

**< > <= >=**

**== !=**

**&&**

**||**

**? :**

Assignment operator: **=** (right to left)

# Operators & Expressions

## Operator Associativity

Kicks in when operators of the same precedence appear in an Expression.

Postfix operators: **++ --** (left to right)

Prefix operators: **++ --** (right to left)

Unary operators: **+ - !** (right to left)

**\* / %** (left to right)

**+ -** (left to right)

**< > <= >=**

**== !=**

**&&**

**||**

**? :**

Assignment operator: **=** (right to left)

Examples with Expressions:

A)  $3 * 6 / 9$   
 $(3 * 6) / 9$   
 $18 / 9$   
 $2$

B) `int x, y, z;`  
`x = y = z = 0;`

# Operators & Expressions

## Operator Associativity

Kicks in when operators of the same precedence appear in an Expression.

Postfix operators: **++ --** (left to right)

Prefix operators: **++ --** (right to left)

Unary operators: **+ - !** (right to left)

**\* / %** (left to right)

**+ -** (left to right)

**< > <= >=**

**== !=**

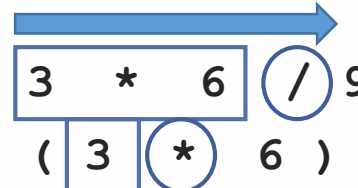
**&&**

**||**

**? :**

Assignment operator: **=** (right to left)

Examples with Expressions:

A)   
3 \* 6 / 9  
( 3 \* 6 ) / 9  
18 / 9  
2

B) `int x, y, z;`  
`x = y = z = 0;`

# Operators & Expressions

## Operator Associativity

Kicks in when operators of the same precedence appear in an Expression.

Postfix operators: **++ --** (left to right)

Prefix operators: **++ --** (right to left)

Unary operators: **+ - !** (right to left)

**\* / %** (left to right)

**+ -** (left to right)

**< > <= >=**

**== !=**

**&&**

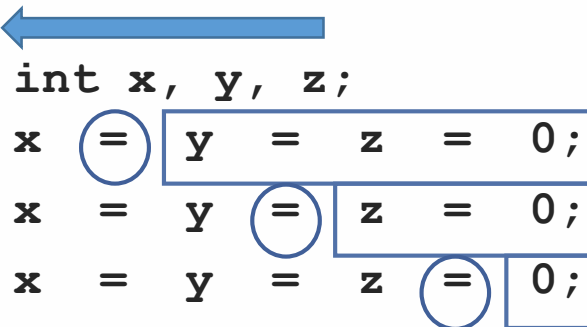
**||**

**? :**

Assignment operator: **=** (right to left)

Examples with Expressions:

A)  $3 * 6 / 9$   
 $(3 * 6) / 9$   
 $18 / 9$   
 $2$

B)   
`int x, y, z;`  
`x = y = z = 0;`  
`x = y = z = 0;`  
`x = y = z = 0;`



# Operators & Expressions

## Operators

Postfix operators: **++ --** (left to right)

Prefix operators: **++ --** (right to left)

Unary operators: **+ - !** (right to left)

**\* / %** (left to right)

**+ -** (left to right)

**< > <= >=**

**== !=**

**&&**

**||**

**? :**

Assignment operator: **=** (right to left)

## Arithmetic precision of calculations

- C++ Rules are a VERY important consideration here !
- Expressions in C++ might not evaluate as you'd "expect"!

“Highest-order operand” determines type of arithmetic “precision”.



# Operators & Expressions

## Operators

Postfix operators: **++** **--** (left to right)

Prefix operators: **++** **--** (right to left)

Unary operators: **+** **-** **!** (right to left)

**\*** **/** **%** (left to right)

**+** **-** (left to right)

**<** **>** **<=** **>=**

**==** **!=**

**&&**

**||**

**?** **:**

Assignment operator: **=** (right to left)

Arithmetic precision of calculations

“Highest-order operand” determines type of arithmetic “precision”.

➤ **17 / 5** evaluates to **3** in C++!

Both operands are **ints**, hence integer division is performed.

➤ **17.0 / 5** evaluates to **3.4** in C++!

Highest-order operand is **double** (**17.0**), hence double precision division is performed.

# Operators & Expressions

## Operators

Postfix operators: **++** **--** (left to right)

Prefix operators: **++** **--** (right to left)

Unary operators: **+** **-** **!** (right to left)

**\*** **/** **%** (left to right)

**+** **-** (left to right)

**<** **>** **<=** **>=**

**==** **!=**

**&&**

**||**

**?** **:**

Assignment operator: **=** (right to left)

Arithmetic precision of calculations

“Highest-order operand” determines type of arithmetic “precision”.

➤ **17 / 5** evaluates to **3** in C++!

Both operands are **ints** : Integer division.

➤ **17.0 / 5** evaluates to **3.4** in C++!

Highest-order operand is **double** : Double division.

➤ **int** intVar1 = 1, intVar2 = 2;  
**double** doubleVar = intVar1 / intVar2;

**doubleVar** is 0.0 !

# Operators & Expressions

## Operators

Postfix operators: **++ --** (left to right)

Prefix operators: **++ --** (right to left)

Unary operators: **+ - !** (right to left)

**\* / %** (left to right)

**+ -** (left to right)

**< > <= >=**

**== !=**

**&&**

**||**

**? :**

Assignment operator: **=** (right to left)

## Arithmetic precision of calculations

“Calculations executed sequentially”

- **1 / 2 / 3.0 / 4** performs 3 separate divisions.  
**(1 / 2)** equals **0**  
**(0 / 3.0)** equals **0.0**  
**(0.0 / 4)** equals **0.0 !**
- “Just one operand” can change the result of a large expression.
- Have to bear in mind all operands & operators rules!

# Operators & Expressions

## Type Casting with ( )• or (•)

Perform explicit type-casting conversion

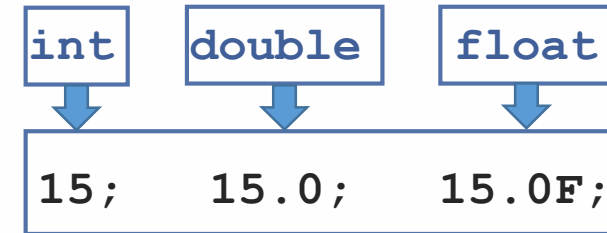
Can add “.0” to literals to force precision:

```
convertedVar = (new_type)originalVar;
```

```
convertedVar = new_type(originalVar);
```

```
double x = (double) intVar1 / intVar2;
```

```
double x = double( intVar1 / intVar2 );
```



Casting to force double-precision division among two integer variables! DOES IT?

Alternative C++ expression:

```
double x = static_cast<double>( x );
```

# Operators & Expressions

## Type Casting with ( )• or (•)

Perform explicit type-casting conversion

Can add “.0” to literals to force precision:     15;     15.0;     15.0F;

```
convertedVar = (new_type)originalVar;
```

```
convertedVar = new_type(originalVar);
```

valid C++ expression

```
double x = (double) intVar1 / intVar2;
```

```
double x = double( intVar1 / intVar2 );
```

Casting to force double-precision division among two integer variables! DOES IT?

Alternative C++ expression:

```
double x = static_cast<double>( X );
```

# Operators & Expressions

## Type Casting Operator ( )• or (•)

Perform explicit type-casting conversion

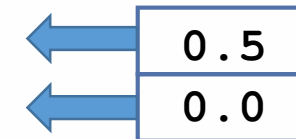
Can add “.0” to literals to force precision:    15;    15.0;    15.0F;

```
convertedVar = (new_type)originalVar;
```

```
convertedVar = new_type(originalVar);
```

```
double x = (double) intVar1 / intVar2;
```

```
double x = double( intVar1 / intVar2 );
```



(For intVar1=1, intVar2=2)

Casting to force double-precision division among two integer variables! DOES IT?

Alternative C++ expression:

```
double x = static_cast<double>( X );
```

# Operators & Expressions

## Type Conversion

- Implicit type conversion

Done by the compiler:

```
17 / 5.5;
```

“Implicit type cast”  $17 \rightarrow 17.0$

- Explicit type conversion

Programmer-enforced:

```
(double)17 / 5.5;
```

```
double(17) / 5.5;
```

```
static_cast<double>( 17 ) / 5.5;
```



# Operators & Expressions

## Shorthand Operators

### ➤ Arithmetic operation & Assignment

EXAMPLE	EQUIVALENT TO
<code>count += 2;</code>	<code>count = count + 2;</code>
<code>total -= discount;</code>	<code>total = total - discount;</code>
<code>bonus *= 2;</code>	<code>bonus = bonus * 2;</code>
<code>time /= rushFactor;</code>	<code>time = time/rushFactor;</code>
<code>change %= 100;</code>	<code>change = change % 100;</code>
<code>amount *= cnt1 + cnt2;</code>	<code>amount = amount * (cnt1 + cnt2);</code>

Also shorthands:

- Post-increment/decrement: `i++` (increment/decrement *then* evaluate expression)
- Pre-increment/decrement: `++i` (evaluate expression *then* increment/decrement)



# Statements

A complete unit of execution (equivalent to a sentence in a language).

➤ Expression statements

Assignment expressions

Use of ( **++** ) or ( **--** )

Method invocations

Object creation

End with semicolon ( ; )

➤ Flow Control statements

Selection structures

Repetition/Iteration structures

Follow Scope rules  
(formally introduced later)

## Flow Control Statements

### ➤ If / then / else

<pre>if (x == 0)     cout &lt;&lt; "0"; cout &lt;&lt; "Done";</pre>	<pre>if (x == 0)     cout &lt;&lt; "0"; else     cout &lt;&lt; "not 0"; cout &lt;&lt; "Done";</pre>
---	---

### Brace-enclosed **Block**

<pre>if (x == 0){     cout &lt;&lt; "x is ";     cout &lt;&lt; "0"; }</pre>
<pre>else{     cout &lt;&lt; "x is ";     cout &lt;&lt; "not 0"; } cout &lt;&lt; "Done";</pre>

Block: a group of zero or more statements that are grouped together by delimiters ( in C++ braces “{” and “}” )

➤ Good practice is to include the curly braces even for single-liners.

# Statements

## Flow Control Statements

➤ If / then / else

```
if (x == 0)
    cout << "0";
    cout << "Done";
```

Note (common error!) :

```
if (x = 0)
    cout << "1";
    cout << "Done";
```

```
if (x == 0)
    cout << "0";
else
    cout << "not 0";
    cout << "Done";
```

Brace-enclosed **Block**

```
if (x == 0) {
    cout << "x is ";
    cout << "0";
}
else {
    cout << "x is ";
    cout << "not 0";
}
    cout << "Done";
```

Block: a group of zero or more statements that are grouped together by delimiters ( in C++ braces “{” and “}” )

➤ Good practice is to include the curly braces even for single-liners.

## Flow Control Statements

### ➤ Switch

- The switching value must evaluate to an integer or enumerated type
- The case values must be either:
  - a) a constant or literal, or
  - b) an **enum** value
- The case values must be of the same type as the switch expression

Notes:

- **break** statements are typically used to terminate each **case**.
- It is usually a good practice to include a **default** case.

```
switch(cardValue) {  
    case 11:  
        cout << "Jack";  
        break;  
    case 12:  
        cout << "Queen";  
        break;  
    case 13:  
        cout << "King";  
        break;  
    default:  
        cout << cardValue;  
        break;  
}
```

## Flow Control Statements

### ➤ Switch

- The switching value must evaluate to an integer or enumerated type
- The case values must be either:
  - a) a constant or literal, or
  - b) an **enum** value
- The case values must be of the same type as the switch expression

Notes:

- **break** statements are typically used to terminate each **case**.
- Without a **break** statement, cases “fall through” to the next statement.

```
switch(cardValue) {  
    case 11:  
        cout << "Jack";  
          
    case 12:  
        cout << "Queen";  
          
    case 13:  
        cout << "King";  
          
    default:  
        cout << cardValue;  
          
}
```

Why?:

- In reality **switch** is like a special kind of **goto** ...
- Means you *should* also brace-enclose each **case** Scope !

## Flow Control Statements

### ➤ While

Executes a block of statements while a particular condition/expression is **true**

```
int count = 0;
while(count < 10) {
    cout << count;
    count++;
}
```

### ➤ Do While

Performs at least one block execution

```
int count = 0;
do {
    cout << count;
    count++;
} while(count < 10)
```

## Flow Control Statements

### ➤ For

Iterate over a range of values.

```
for ( init; term; incr ) {  
    ...  
}
```

- The *initialization* expression initializes the loop it is executed once, as the loop begins.
- Loop ends when the *termination* expression evaluates to **false**.
- The *increment* expression is invoked after each iteration.

```
for (int count = 0; count < 10; count++) {  
    cout << count;  
}
```

```
for (int count = 25; count < 50; count += 5){ //increment by 5  
    cout << count;  
}
```



## Flow Control Statements

### ➤ For

Iterate over a range of values.

```
for ( init; term; incr ) {  
    ...  
}
```

- The *initialization* expression initializes the loop it is executed once, as the loop begins.
- Loop ends when the *termination* expression evaluates to **false**.
- The *increment* expression is invoked after each iteration.

```
for ( ; ; ) {  
    cout << "Running" << endl;  
}
```

//continuously running, no increment

```
for (int count = 0 ; ; ++count){  
    cout << count << endl;  
}
```

//continuously running, increment by 1



# Input / Output

```
1  #include <iostream>
2  using namespace std;

3  int main( )
4  {
5      int numberOfLanguages;

6      cout << "Hello reader.\n"
7           << "Welcome to C++.\n";

8      cout << "How many programming languages have you used? ";
9      cin >> numberOfLanguages;

10     if (numberOfLanguages < 1)
11         cout << "Read the preface. You may prefer\n"
12              << "a more elementary book by the same author.\n";
13     else
14         cout << "Enjoy the book.\n";

15     return 0;
16 }
```

Console  
Input / Output

# Input / Output

## Console Input / Output

- Console Input, Output, and Error stream objects in C++ are called:  
`cin`, `cout`, `cerr`
- They are Global Objects of the classes  
`ostream` (outputstream) and `istream` (inputstream)
- Defined in the C++ library header called `<iostream>`  
(we'll leave it at that for now)

Useful for:

- User input
- User output
- Error messages (exclusive stream, redirection if required)

# Input / Output

```
1  #include <iostream>
2  using namespace std;

3  int main( )
4  {
5      int numberOfLanguages;

6      cout << "Hello reader.\n"
7           << "Welcome to C++.\n";

8      cout << "How many programming languages have you used? ";
9      cin >> numberOfLanguages;

10     if (numberOfLanguages < 1)
11         cout << "Read the preface. You may prefer\n"
12              << "a more elementary book by the same author.\n";
13     else
14         cout << "Enjoy the book.\n";

15     return 0;
16 }
```

## Preprocessor directives

Note:

**using namespace std;**

Without it:

**std::cout**

**std::cin**

**std::cerr**

# Input / Output

## Console Input / Output

- Console Input, Output, and Error stream objects in C++ are called:  
**cin**, **cout**, **cerr**
- They are Global Objects of the classes  
**ostream** (outputstream) and **istream** (inputstream)
- Defined in the C++ library called **<iostream>**

Side-Note:

Guaranteed at least past **C++11**

*Note:*

**std::cout** and **std::cin** are Global Objects of the classes **std::ostream** and **std::istream**

- **#include <iostream>** is responsible for including their corresponding declarations in your programs.

# Input / Output

## Console Output ( `std::cout` )

Any standard C++ data can be output:

- Variables
- Constants
- Literals
- Expressions (which can include all of above)

```
cout << numberOfGames << " games played.";
```

2 values are output:

Value of variable `numberOfGames`

Literal string `" games played."`

- Cascading: multiple values with one `cout`-initiated expression.

Note:  
Insertion Operators

# Input / Output

## Output

New lines in output

- Escape sequences are valid: `"\n"` is “newline”

A second method:

- Object `std::endl`
- Flushes output buffer ( `std::flush` )

Examples:

```
cout << "Hello World\n";
```

```
cout << "Hello World" << endl;
```

SEQUENCE	MEANING
<code>\n</code>	New line
<code>\r</code>	Carriage return (Positions the cursor at the start of the current line. You are not likely to use this very much.)
<code>\t</code>	(Horizontal) Tab (Advances the cursor to the next tab stop.)
<code>\a</code>	Alert (Sounds the alert noise, typically a bell.)
<code>\\</code>	Backslash (Allows you to place a backslash in a quoted expression.)
<code>\'</code>	Single quote (Mostly used to place a single quote inside single quotes.)
<code>\"</code>	Double quote (Mostly used to place a double quote inside a quoted string.)

- Makes sense to *force output* of heavy, crash-prone processes.
- Creates overhead.
- Same in line-buffered context.



# Input / Output

## Output Format

Numeric values may not display as you'd expect:

```
cout << "The price is $" << price << endl;
```

If `double price = 78.5;` we might get:

The price is \$78.500000

The price is \$78.5

➤ Force Decimals:

```
cout.setf(ios::fixed);  
cout.setf(ios::showpoint);  
cout.precision(2);
```

Fixed Precision  
Show Decimal Point  
Set Precision Decimals

Side-Note:  
Guaranteed at least past C++11

*Note:*

`std::cout` and `std::cin` are Global Objects of the classes `std::ostream` and `std::istream`  
With their corresponding class *methods* `setf()` and `precision()`  
e.g. `cout.setf(...);` and / or `cin.setf(...);` you can change their “attributes”.

```
cout.precision(...);  
cin.precision(...);
```



# Input / Output

## Console Input ( `std::cin` )

No literals allowed for `cin`

- Must input to a variable

Waits on-screen for keyboard entry

- `cin >> num;`

Value entered at keyboard is ‘assigned’ to `num`.

`cin` `>>` `firstName` `>>` `lastName` `>>` `age`;

Note:  
**Extraction Operators**

- Consumes any leading whitespaces, and stops reading at next whitespace.
- Can also be cascaded, `>>` operators separate each “type” of thing we read in.

# Input / Output

## Console Input ( `std::cin` )

No literals allowed for `cin`

- Must input to a variable

Waits on-screen for keyboard entry

- `cin >> num;`

Value entered at keyboard is ‘assigned’ to `num`.

`cin >> firstName >> lastName >> age;`

- Consumes any leading whitespaces, and stops reading at next whitespace.

Example type-in: [ws][ws]420[ws]911[↵]

`num : 420`

- Can also be cascaded, `>>` operators separate each “type” of thing we read in.

# Input / Output

## Console Input ( `std::cin` )

No literals allowed for `cin`

- Must input to a variable

Waits on-screen for keyboard entry

- `cin >> num;`

Value entered at keyboard is ‘assigned’ to `num`.

`cin >> firstName >> lastName >> age;`

- Consumes any leading whitespaces, and stops reading at next whitespace.

Example type-in:    [ws][ws] **420** [ws] **911** [↵]                    **num : 420**

- Can also be cascaded, `>>` operators separate each “type” of thing we read in.

Example type-in:    **christos** [ws] **papachristos** [ws] **33** [↵]

# Input / Output

## User Input /Output

Prompt user for input

```
cout << "Enter number of objects: ";  
cin >> numObjects;
```

### Note:

no `"\n"` or `std::endl` in `cout` here.  
Prompt will “wait” for user input on the same line !

User-friendly input/output design:

- Every `cin` should have a corresponding prior `cout` prompt.

# Input / Output

## User Input /Output

Prompt user for input

```
1  //Program to demonstrate cin and cout
2  #include <iostream>
3  #include <string>

4  using namespace std;
5  int main( )
6  {
7      string dogName;
8      int actualAge;
9      int humanAge;

10     cout << "How many years old is your dog?" << endl;
11     cin >> actualAge;
12     humanAge = actualAge * 7;

13     cout << "What is your dog's name?" << endl;
14     cin >> dogName;

15     cout << dogName << "'s age is approximately " <<
16         "equivalent to a " << humanAge << " year old human."
17         << endl;

18     return 0;
19 }
```

# Input / Output

## User Input / Output

Whitespace Behavior:

### Note:

Will stop at whitespace.

```
cin >> dogName;  
cout << dogName;
```

Mr .

or

We could have done instead:

```
cin >> dogTitle  
    >> dogName;
```

### Sample Dialogue 1

How many years old is your dog?

5

What is your dog's name?

Rex

Rex's age is approximately equivalent to a 35 year old human.

### Sample Dialogue 2

How many years old is your dog?

10

What is your dog's name?

Mr. Bojangles

Mr.'s age is approximately equivalent to a 70 year old human.

*"Bojangles" is not read into  
dogName because cin stops  
input at the space.*

# Input / Output

## User Input / Output

Whitespace Skipping,  
an Example:

Note:

`std::noskipws`  
`std::skipws`

`cin << skipws << ... ;`

or

`cin.setf( skipws );`  
`cin << ... ;`

`cin << noskipws << ... ;`

or

`cin.setf( noskipws );`  
`cin << ... ;`

Note: Default is to **skipws**.

```
#include <iostream>

int main () {
    char a, b, c;

    // set whitespace skip flag, read in " 0 1 2"
    std::cin >> std::skipws >> a >> b >> c;
    std::cout << a << "," << b << "," << c << std::endl;

    // flushes cin
    std::cin.ignore(INT_MAX);

    // unset whitespace skip flag, read in " 0 1 2"
    std::cin >> std::noskipws >> a >> b >> c;
    std::cout << a << "," << b << "," << c << std::endl;

    return 0;
}
```



# Input / Output

## User Input / Output

Whitespace Skipping,  
another Example:

**Note:**

`std::noskipws`  
`std::skipws`

`cin << skipws << ... ;`

or

`cin.setf( skipws );`  
`cin << ... ;`

`cin << noskipws << ... ;`

or

`cin.setf( noskipws );`  
`cin << ... ;`

**Note:** Default is to `skipws`.

```
#include <iostream>
```

```
int main () {
```

```
    char a, b, c;
```

```
    // set whitespace skip flag, read in " 0 1 2"
    std::cin >> std::skipws >> a >> b >> c;
    std::cout << a << "," << b << "," << c << std::endl;
```

```
Input:  [ws][ws] 0 [ws][ws][ws] 1 [ws] 2
Output: 0,1,2
```

```
    // flushes cin
```

```
    std::cin.ignore(INT_MAX);
```

```
    // unset whitespace skip flag, read in " 0 1 2"
    std::cin >> std::noskipws >> a >> b >> c;
    std::cout << a << "," << b << "," << c << std::endl;
```

```
    return 0;
```

```
}
```

# Input / Output

## User Input / Output

Whitespace Skipping,  
another Example:

Note:

`std::noskipws`  
`std::skipws`

`cin << skipws << ... ;`

or

`cin.setf( skipws );`  
`cin << ... ;`

`cin << noskipws << ... ;`

or

`cin.setf( noskipws );`  
`cin << ... ;`

Note: Default is to `skipws`.

```
#include <iostream>
```

```
int main () {
```

```
    char a, b, c;
```

```
    // set whitespace skip flag, read in " 0 1 2"
```

```
    std::cin >> std::skipws >> a >> b >> c;
```

```
    std::cout << a << "," << b << "," << c << std::endl;
```

```
    // flushes cin
```

```
    std::cin.ignore(INT_MAX);
```

```
    // unset whitespace skip flag, read in " 0 1 2"
```

```
    std::cin >> std::noskipws >> a >> b >> c;
```

```
    std::cout << a << "," << b << "," << c << std::endl;
```

Input:      [ws][ws] 0 [ws][ws][ws] 1 [ws] 2

Output:     [ws],[ws],0

```
    return 0;
```

```
}
```

# Input / Output

## User Input / Output

Another solution:

Read-in the entire line !

**Note:**

```
getline ( char* s,  
          streamsize n,  
          char delim );
```

Takes a C-string (**char** array)  
to store result,  
the size of the C-string array,  
and a delimiting **char**  
(default is **'\n'**)

```
#include <iostream>  
  
const int STR_SIZE = 256;  
  
int main () {  
    char in_str[STR_SIZE];  
  
    // without whitespace skip flag read in " 10 1 23"  
    // by getting the entire line  
    std::cin.getline(in_str, STR_SIZE);  
    std::cout << in_str << std::endl;  
    std::cout << atoi(in_str) << std::endl;  
    std::cout << atoi(&in_str[2]) << ","  
                << atoi(&in_str[7]) << ","  
                << atoi(&in_str[9]) << std::endl;  
  
    return 0;  
}
```

```
Input:  [ws][ws] 10 [ws][ws][ws] 1 [ws] 23  
Output: [ws][ws] 10 [ws][ws][ws] 1 [ws] 23
```

# Input / Output

## User Input / Output

Read-in the entire line,  
another Solution:

**Note:**

```
getline ( char* s,  
          streamsize n,  
          char delim );
```

Takes a C-string (**char** array)  
to store result,  
the size of the C-string array,  
and a delimiting **char**  
(default is **'\n'**)

```
#include <iostream>  
  
const int STR_SIZE = 256;  
  
int main () {  
    char in_str[STR_SIZE];  
  
    // without whitespace skip flag read in " 10  1 23"  
    // by getting the entire line  
    std::cin.getline(in_str, STR_SIZE);  
    std::cout << in_str << std::endl;  
    std::cout << atoi(in_str) << std::endl;  
    std::cout << atoi(&in_str[2]) << ","  
                << atoi(&in_str[7]) << ","  
                << atoi(&in_str[9]) << std::endl;  
  
    return 0;  
}
```

Needs  
Parsing

```
Input:  [ws][ws] 10 [ws][ws][ws] 1 [ws] 23  
Output: 10
```

# Input / Output

## User Input / Output

Read-in the entire line,  
another Solution:

**Note:**

```
getline ( char* s,  
          streamsize n,  
          char delim );
```

Takes a C-string (**char** array)  
to store result,  
the size of the C-string array,  
and a delimiting **char**  
(default is **'\n'**)

```
#include <iostream>  
  
const int STR_SIZE = 256;  
  
int main () {  
  
    char in_str[STR_SIZE];  
  
    // without whitespace skip flag read in " 10  1 23"  
    // by getting the entire line  
    std::cin.getline(in_str, STR_SIZE);  
    std::cout << in_str << std::endl;  
    std::cout << atoi(in_str) << std::endl;  
  
    std::cout << atoi(&in_str[2]) << ","  
               << atoi(&in_str[7]) << ","  
               << atoi(&in_str[9]) << std::endl;  
  
    return 0;  
}
```

But HOW?

```
Input:  [ws][ws] 10 [ws][ws][ws] 1 [ws] 23  
Output: 10,1,23
```

## Error Output ( `std::cerr` )

`cerr` works same as `cout`

- Mechanism for distinguishing between regular output and error output
- Most systems allow `cout` and `cerr` to be “redirected” to other devices e.g., line printer, output file, error console, etc.

# Input / Output

## File Input / Output

Similarly to `cin`, a combination of:

➤ `cin >> num;`

Input Object (C++)  
Extraction Operator  
Variables

At the top:

```
#include <fstream>
using namespace std;
```

An input stream object (creation just as with any other variable):

```
ifstream inputStream;
```

“Connect” the `inputStream` variable to a text file (via pathname):

```
inputStream.open("filename.txt");
```



# Input / Output

## File Input / Output

Read-in by using the Extraction Operator ( `>>` ):

```
InputStream >> var;
```

The result is the same as using `cin >> var` except the input is coming from the text file and not the keyboard.

Check that EOF hasn't been reached:

```
if ( !InputStream.eof() )
```

Close with :

```
InputStream.close();
```

# Input / Output

## File Input / Output

Output (similarly):

An output stream object (creation just as with any other variable):

```
ofstream outputStream;
```

Open file to write:

or

```
outputStream.open("filename.txt", ofstream::out);  
outputStream.open("filename.txt");
```

Write-out by using the Insertion Operator ( << ):

```
outputStream << var;
```

Close with :

```
outputStream.close();
```

# Input / Output

## File Input / Output

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>

4  using namespace std;
5  int main( )
6  {
7      string firstName, lastName;
8      int score;
9      fstream inputStream;

10     inputStream.open("player.txt");

11     inputStream >> score;
12     inputStream >> firstName >> lastName;

13     cout << "Name: " << firstName << " "
14           << lastName << endl;
15     cout << "Score: " << score << endl;
16     inputStream.close();

17     return 0;
18 }
```

player.txt

100510  
Gordon Freeman

### Sample Dialogue

Name: Gordon Freeman  
Score: 100510

# Namespaces - Resolution

## Namespaces

A collection of name definitions under a **top-level identifier**.

Most common is **namespace std**

- Contains *all* standard library definitions !

The **using** keyword: Instruct the compiler to attempt to resolve names therein

Examples:

```
#include <iostream>
using namespace std;
```

or

```
#include <iostream>
using std::cin;
using std::cout;
```

What is the difference ?

# Namespaces - Resolution

## Namespaces

A collection of name definitions under a **top-level identifier**.

Most common is **namespace std**

- Contains *all* standard library definitions !

The **using** keyword: Instruct the compiler to attempt to resolve names therein

Examples:

```
#include <iostream>
using namespace std;
```

or

```
#include <iostream>
using std::cin;
using std::cout;
```

Includes entire standard library of  
name definitions:  
**cout** , **cin** , **cerr** , **endl**

# Namespaces - Resolution

## Namespaces

A collection of name definitions under a **top-level identifier**.

Most common is **namespace std**

- Contains *all* standard library definitions !

The **using** keyword: Instruct the compiler to attempt to resolve names therein

Examples:

```
#include <iostream>
using namespace std;
```

or

```
#include <iostream>
using std::cin;
using std::cout;
```

Includes entire standard library of  
name definitions:

**cout** , **cin** , **cerr** , **endl**

Specify just the objects we want

# Namespaces - Resolution

## Resolution Operator ( :: )

Explicit resolution under a **namespace**

Objects: `std::cout`

Functions: `std::count( its, itl, val )`

In case of name conflicts, it *might* supersede any **using** keyword usage:

```
#include <iostream>
using namespace std;
```

```
namespace ns{
    ...
    int cout = 1; Namespace declaration
    ...
}
```

```
...
cout << ns::cout;
```



# Namespaces - Resolution

## Resolution Operator ( :: )

Explicit resolution under a **namespace**

Objects: `std::cout`

Functions: `std::count( its, itl, val )`

In case of name conflicts, it *might* supersede any **using** keyword usage:

```
#include <iostream>
using namespace std;
```

```
namespace ns{
    ...
    int cout = 1; Namespace declaration
    ...
}
```

```
...
cout << ns::cout;
```

- `cout` evaluates to `std::cout`
- `ns::cout` evaluates to the variable in `ns`

# Scope

You can define new variables in many places in your code.  
So where is it in effect / What is its Variable Scope?

➤ The set of statements in which the variable is known to the compiler.

Where a variable can be referenced from in your program

➤ Limited by the code **Block** in which the variable is defined

```
if(age >= 18) {  
    bool adult = true;  
    cout << adult;  
}  
cout << adult;
```

```
bool adult = false;  
if(age >= 18) {  
    bool adult = true;  
    cout << adult;  
}  
cout << adult;
```

# Scope

You can define new variables in many places in your code.  
So where is it in effect / What is its Variable Scope?

➤ The set of statements in which the variable is known to the compiler.

Where a variable can be referenced from in your program

➤ Limited by the code **Block** in which the variable is defined

```
if(age >= 18) {  
    bool adult = true;  
    cout << adult;  
}  
cout << adult;
```

```
bool adult = false;  
if(age >= 18) {  
    bool adult = true;  
    cout << adult;  
}  
cout << adult;
```

# Scope

You can define new variables in many places in your code.  
So where is it in effect / What is its Variable Scope?

- The set of statements in which the variable is known to the compiler.

Where a variable can be referenced from in your program

- Limited by the code **Block** in which the variable is defined

```
if(age >= 18) {  
    bool adult = true;  
    cout << adult;  
}  
cout << adult;
```

```
bool adult = false;  
if(age >= 18) {  
    bool adult = true;  
    cout << adult;  
}  
cout << adult;
```

**The Block Scope { }**  
(it's more generic)

```
bool adult = false;  
{  
    bool adult = true;  
    cout << adult;  
}  
cout << adult;
```

## Scope Resolution (Ambiguities)

Revisiting the (BAD!) practice of `using namespace std;`

Functions: `std::count( its, itl, val)`

```
#include <algorithm>
using namespace std;
```

```
int count = 0;
int increment() {
    return ++count;
}
```

```
#include <algorithm>
using namespace std;
```

```
int increment() {
    int count = 0;
    return ++count;
}
```

Long error code...

```
error: reference to 'count' is ambiguous: note: candidates are: int count In file included from
/usr/include/c++/4.9/algorithm:62:0, from 2: /usr/include/c++/4.9/bits/stl_algo.h:3947:5: note: template<class
_IIter, class _Tp> typename std::iterator_traits<_Iterator>::difference_type std::count(_IIter, _IIter, const
_Tp&) count(_InputIterator __first, _InputIterator __last, const _Tp& __value)
```

# Scope

## Scope Resolution (Ambiguities)

Revisiting the (BAD!) practice of `using namespace std;`

Functions: `std::count( its, itl, val)`

Why?

```
#include <algorithm>

int increment() {
    using namespace std;
    int count = 0;
    return ++count;
}
```

```
#include <algorithm>
int count = 0;

int increment() {
    using namespace std;
    return ++count;
}
```

Ambiguous

# Scope

## Scope Resolution (Ambiguities)

The (BAD!) practice of `using namespace std;`  
Functions: `std::count( its, itl, val)`

Rule looks at Global Scope

- Behaves “as-if” it’s placed together with `#include` statements, even though it’s trying to import names into the Local Scope only.

```
#include <algorithm>

int increment() {
    using namespace std;
    int count = 0;
    return ++count;
}
```

```
#include <algorithm>
int count = 0;

int increment() {
    using namespace std;
    return ++count;
}
```

Ambiguous

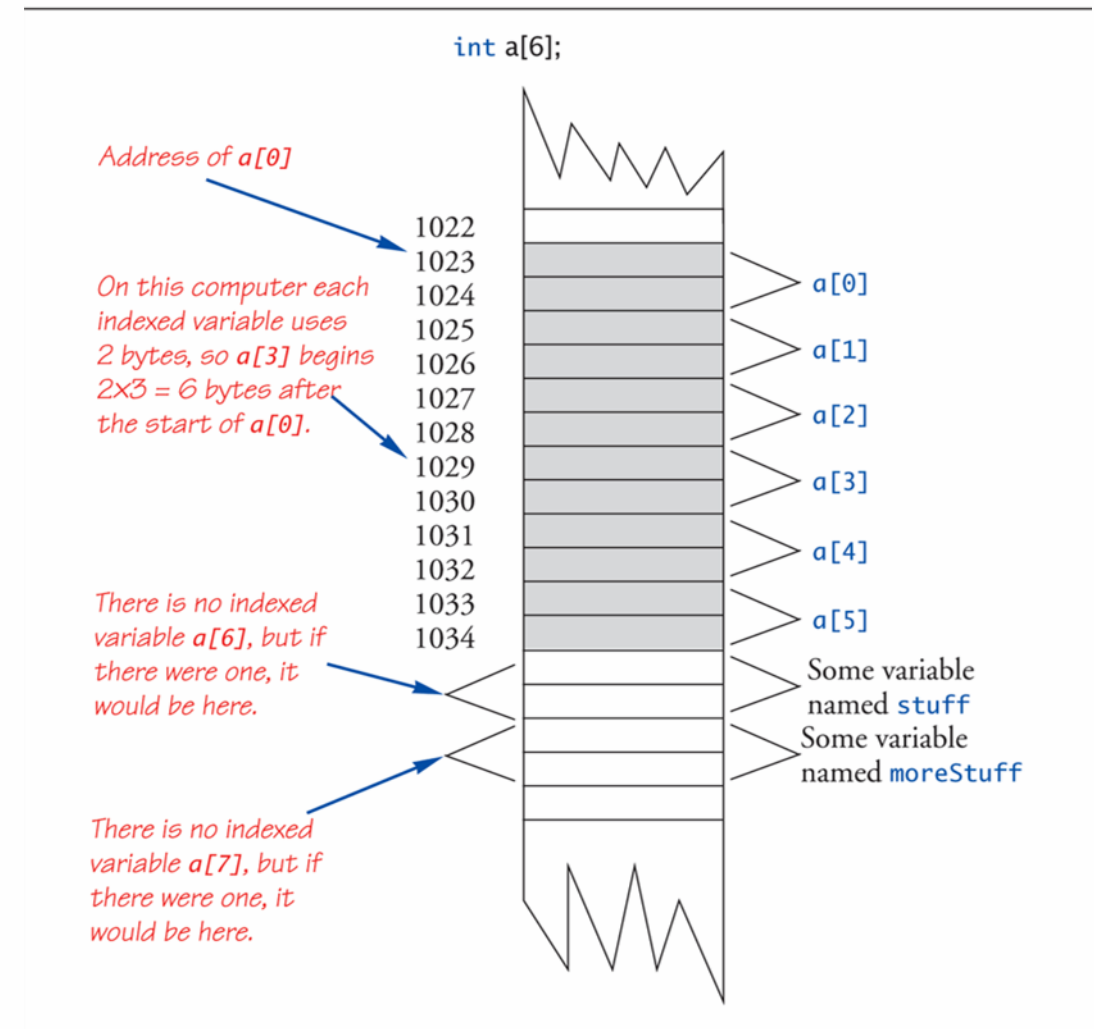


# Arrays

A collection of related data items.

- Can be of any data type.
- They are static  
Their size does not change.

They are declared contiguously in memory.  
In other words, an array's data is stored in one big block, together.



# Arrays

Recall simple variables:

- Allocated memory in an "address"

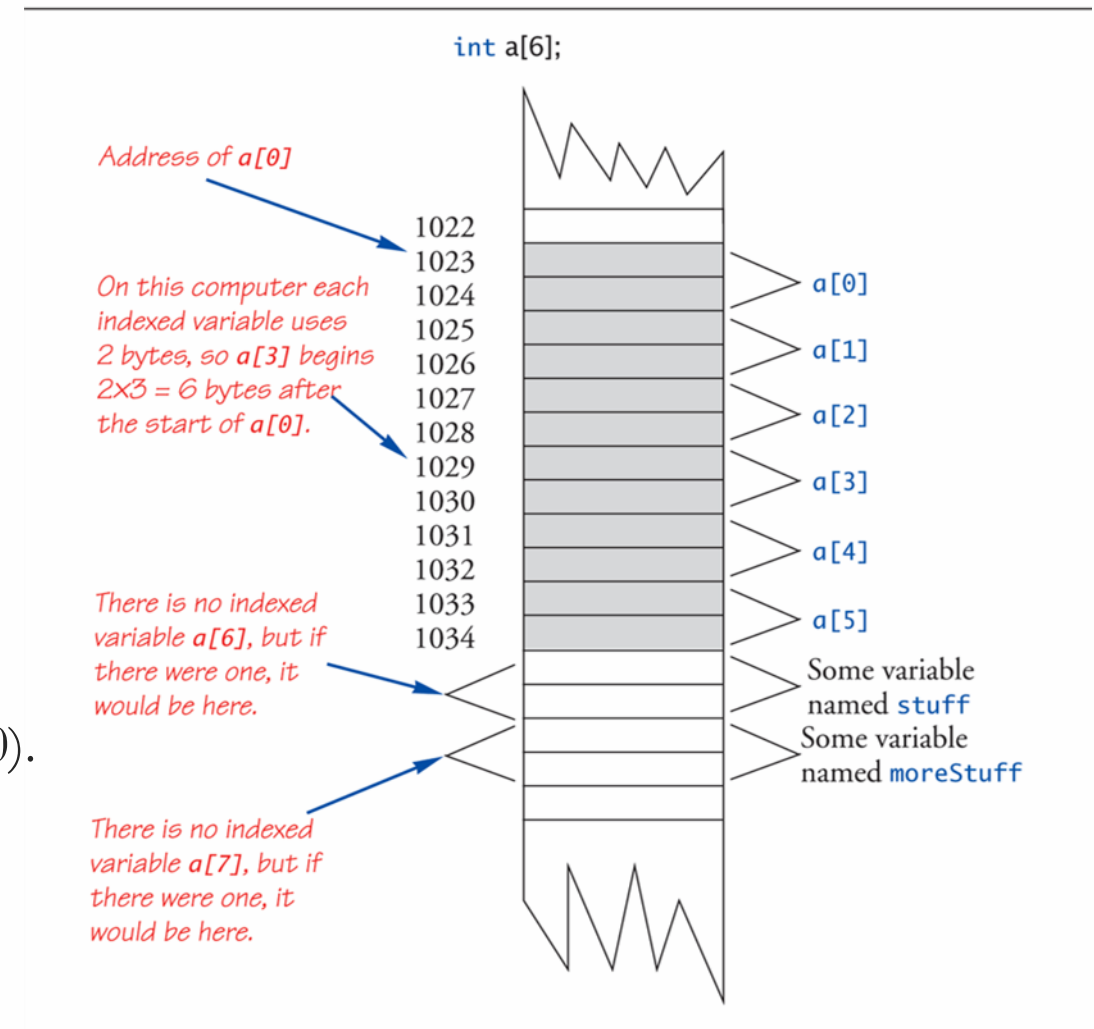
Array declarations allocate memory for entire array

- Sequential allocation

Addresses allocated "back-to-back".

Allows indexing calculations.

Simple "addition" from array beginning (index 0).



## Array Declaration

```
<type> <name> [size];  
float xArray [10];
```

This array now has memory to hold **size=10** floats.

0-based indexing (0 is our natural “first” number):

**xArray[9];** At **size-1** lies the final element of the array.

C++ pitfall:

The compiler will “let you go” beyond **size-1**.

Compiler will not detect this as an error.

```
xArray[10] = 1.0F;
```

Unpredictable results! Up to programmer to “stay in range”.

## Array Limitations

- Does not know how large it is – there is no C++ `size()` function for arrays.
- No bounds checking is performed.

## Arrays are static

- Size must be known at compile time (cannot change once set).  
*Normally* can't do user input for array size: “How many numbers would you like to store?”

## C / C++ Benefits:

- Efficiency.
- Backwards Compatibility.

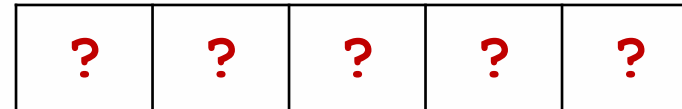
# Arrays

## Array Declaration / Initialization

- A *declaration* alone generally will not initialize the data stored in the memory locations.
- They will contain “garbage” leftover data.

Declaration:

```
int numbers[5];
```



Allocates array **yArray** to hold 3 integers

## Array Declaration / Initialization

➤ *Initialization* ensures specific values for the contained data.

Declaration - initialization:

```
int numbers[5] = { 5, 2, 6, 9, 3 };
```

5	2	6	9	3
---	---	---	---	---

Allocates array **yArray** to hold 3 integers

## Array Declaration / Initialization

- *Initialization* ensures specific values for the contained data.

Declaration - initialization:

```
int numbers[5] = { 5, 2, 6 };
```

5	2	6	0	0
---	---	---	---	---

Auto – initialization (fewer values than the given size) :

- Fills values starting at the beginning.
- Remainder is filled with that data type’s “zero”.



## Array Declaration / Initialization

If no array size is given array is created only as big as is needed:

```
int yArray[] = { 5, 12, 11 };   Allocates array yArray to hold 3 integers
```

# Arrays

## C-strings (as `char` Arrays)

➤ They are `char` type arrays.

➤ Initialization (normal way):

```
char name[5] = {'J', 'o', 'h', 'n', 0};
```

➤ Initialization (string constant literal):

```
char name[5] = "John";
```

NULL-char delimited !

Note: Different quotes have different purposes !!!

➤ Double quotes are for strings

➤ Single quotes are for chars (characters)

# Arrays

## Array Element Access

Bracket Operator ( [•] ):

- Access of a single element (when used on existing instance).

0	1	2	3	4
5	2	6	9	3

```
int numbers[5] = { 5, 2, 6, 9, 3 };  
cout << " The third element is" << numbers[2] << endl;
```

Output:

The third element is 6

# Arrays

## Array Element Access

- C++ also accepts any expression as a “size”  
(must evaluate to an integral value, based on values also known at compile-time).

```
const int start = 0, end = 4;  
double dNumbers[(start + end) / 2];
```

## Array Size using Constants

- Use defined/named constants for your size.

```
or #define NUMBER_OF_STUDENTS 5  
const int NUMBER_OF_STUDENTS = 5;
```

```
int score[NUMBER_OF_STUDENTS];
```

Readability, Versatility, Maintainability

## Array Element Access

- And a non-Standard extension by GCC

```
const int start, end;  
...  
double dNumbers[(start + end) / 2];
```

By the GNU Compiler Collection – Online Docs  
( <http://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html> )

- Variable-length automatic arrays are allowed in ISO C99, and as an extension GCC accepts them in C90 mode and in C++. These arrays are declared like any other automatic arrays, but with a length that is not a constant expression. The storage is allocated at the point of declaration and deallocated when the block scope containing the declaration exits.

*Note:* Make sure you initialize these, otherwise you might never notice a problem until it's too late!

```
int start = 0, end = 100;  
...  
cin >> start >> end;  
...  
double dNumbers[(start + end) / 2];  
dNumbers[0] = -1.0;  
cout << (start+end)/2 << ", " <<  
dNumbers[0] << ", " << dNumbers[100];
```

# Arrays

## Arrays (as Arguments in Functions)

- Indexed variables (individual element of an array is passed):

Function declaration:

```
void myFunction(double param1);
```

Variables:

```
double n, a[10];
```

Function calls:

```
myFunction( a[3] );  
myFunction( n );
```

A **double** in both cases

## Arrays (as Arguments in Functions)

- Entire arrays (passed by the array's name)  
Must pass **size** of array as well, done as second parameter of **int**-type.

### SAMPLE DIALOGUEFUNCTION DECLARATION

```
void fillUp(int a[], int size);
```

### SAMPLE DIALOGUEFUNCTION DEFINITION

```
void fillUp(int a[], int size)
{
    cout << "Enter " << size << " numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> a[i];
    cout << "The last array index used is " << (size - 1) << endl;
}
```



# Arrays

## Arrays (as Arguments in Functions)

- Entire arrays (passed by the array's name)

Example code inside a program `main()`:

```
void fillUp( int a[], int size);
```

Brackets in function definition.

```
int score[5], numberOfScores = 5;  
fillUp(score, numberOfScores);
```

Brackets in variable declaration.

No brackets when passing!

- How does this work? What's really passed?  
*Address-Of* first indexed variable ( `arrName[0]` ).

## Multi-Dimensional Arrays

- Arrays with more than one index

```
char array2d [DIM2] [DIM1];  
char page [30] [100];
```

- Two indices (it is an “*array of arrays*”)

```
page[0][0], page[0][1], ..., page[0][99]  
page[1][0], page[1][1], ..., page[1][99]  
...  
page[29][0], page[29][1], ..., page[29][99]
```

- C++ allows any number of indexes  
Typically no more than two or three

## Multi-Dimensional Arrays

- Arrays with more than one index  
`char array2d [DIM2] [DIM1];`  
`char page [30] [100];`

COLS

- Two indices (it is an “*array of arrays*”)

`page[0][0], page[0][1], ..., page[0][99]`

`page[1][0], page[1][1], ..., page[1][99]`

...

`page[29][0], page[29][1], ..., page[29][99]`

- C++ allows any number of indexes  
Typically no more than two or three

## Multi-Dimensional Arrays

- Arrays with more than one index

```
char array2d [DIM2] [DIM1];  
char page [30] [100];
```

ROWS

COLS

- Two indices (it is an “*array of arrays*”)

```
page[0][0], page[0][1], ..., page[0][99]
```

```
page[1][0], page[1][1], ..., page[1][99]
```

```
...
```

```
page[29][0], page[29][1], ..., page[29][99]
```

- C++ allows any number of indexes  
Typically no more than two or three

# Arrays

## Multi-Dimensional Arrays

- Arrays with more than one index

```
char array2d [DIM2][DIM1];  
char page [30][100];
```

ROWS

COLS

- Two indices (it is an “*array of arrays*”)

```
page[0][0], page[0][1], ..., page[0][99]
```

```
page[1][0], page[1][1], ..., page[1][99]
```

...

```
page[29][0], page[29][1], ..., page[29][99]
```

- C++ allows any number of indexes  
Typically no more than two or three

Array of Arrays

## Multi-Dimensional Arrays

- Indexing with Bracket Operator ( [•] )

```
char a = array2d [j][i];
```

- Multi-Dimensional Arrays as Parameters (Similar to one-dimensional array)

1st dimension size not given (#ROWS), provided as second parameter of function

2nd dimension size is given (#COLS)

```
void DisplayPage(char page[][100], int numRows) {  
    for ( int i = 0; i < numRows; i++ ) {  
        for ( int j = 0; j < 100; j++ ) {  
            cout << page[i][j];  
        }  
        cout << endl;  
    }  
}
```

*Note:*

Otherwise, **error**: declaration of 'page' as multidimensional array must have bounds for all dimensions except the first.

## Multi-Dimensional Arrays

- Indexing with Bracket Operator ( [•] )

```
char a = array2d [j][i];
```

- Multi-Dimensional Arrays as Parameters (Similar to one-dimensional array)

1st dimension size not given (#ROWS) provided as second parameter of function

2nd dimension size is given (#COLS)

```
void DisplayPage(char page[][100], int numRows) {  
    for ( int i = 0; i < numRows; i++ ) {  
        for ( int j = 0; j < 100; j++ ) {  
            cout << page[i][j];  
        }  
        cout << endl;  
    }  
}
```

*Note:*

In fact the declared parameter type is interpreted as **char (\*) [100] !**



## Multi-Dimensional Arrays

- Indexing with Bracket Operator ( [•] )

```
char a = array2d [j][i];
```

- Multi-Dimensional Arrays as Parameters (Similar to one-dimensional array)

1st dimension size not given (#ROWS) provided as second parameter of function

2nd dimension size is given (#COLS)

```
void DisplayPage(char page[][100], int numRows) {  
    for ( int i = 0; i < numRows; i++ ) {  
        for ( int j = 0; j < 100; j++ ) {  
            cout << page[i][j];  
        }  
        cout << endl;  
    }  
}
```

*Note:*

In fact the declared parameter type is interpreted as **char (\*) [100] !**



**CS-202**

Time for Questions !