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	**Project DEADLINE**	NEW Project	

Your 1st Lab is today Thursday 1/24.

Your 1st 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	::	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	

Precedence	Operator	Description	Associativity
10	== !=	For relational operators = and ≠ respectively	Left-to-right
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

Source:
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`)

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.

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`

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 ;
```

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 ;
```

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 ;
```

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 ;
```

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

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 2nd Expression

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 `int`s, 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 `int`s : 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:

`int`	`double`	`float`
`15;`	`15.0;`	`15.0F;`

```
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 );
```

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;      0.5
double x = double( intVar1 / intVar2 );     0.0
```

(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 );
```

Type Conversion

- Implicit type conversion

  Done by the compiler:

  ```
  17 / 5.5;
  ```

  "Implicit type cast" `17` → `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
`count += 2;`	`count = count + 2;`
`total -= discount;`	`total = total - discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rushFactor;`	`time = time/rushFactor;`
`change %= 100;`	`change = change % 100;`
`amount *= cnt1 + cnt2;`	`amount = amount * (cnt1 + cnt2);`

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

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

- If / then / else

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

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

Note (common error!) :

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

```
switch(cardValue) {
 case 11:
  cout << "Jack";

 case 12:
  cout << "Queen";

 case 13:
  cout << "King";

 default:
  cout << cardValue;

}
```

Notes:

• **break** statements are typically used to terminate each **case**.

• Without a **break** statement, cases "fall through" to the next statement.

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

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

int main( )
{
    int numberOfLanguages;

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

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

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

    return 0;
}
```

Console
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

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

Preprocessor directives

```cpp
int main( )
{
    int numberOfLanguages;

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

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

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

    return 0;
}
```

Note:

`using namespace std;`

Without it:

`std::cout`
`std::cin`
`std::cerr`

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.

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."`

Note:
Insertion Operators

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

Input / Output

Output

New lines in output

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

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

A second method:

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

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

Examples:

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

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

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(…);` `cout.precision(…);` and / or `cin.setf(…);` `cin.precision(…);` you can change their "attributes".

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.

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.

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` $\boxed{\texttt{>>}}$ `firstName` $\boxed{\texttt{>>}}$ `lastName` $\boxed{\texttt{>>}}$ `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** [↵]

User Input /Output

Prompt user for input

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

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

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

using namespace std;
int main( )
{
  string dogName;
  int actualAge;
  int humanAge;

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

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

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

  return 0;
}
```

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.*

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;
}
```

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;
}
```

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

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.

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");
```

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();
```

File Input / Output

Output (similarly):

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

```
ofstream outputStream;
```

Open file to write:

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

or

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

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

```
outputStream << var;
```

Close with :

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

Input / Output

File Input / Output

```cpp
#include <iostream>
#include <fstream>
#include <string>

using namespace std;
int main( )
{
    string firstName, lastName;
    int score;
    fstream inputStream;

    inputStream.open("player.txt");

    inputStream >> score;
    inputStream >> firstName >> lastName;

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

    return 0;
}
```

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;
```

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

or

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

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

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

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

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

```
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 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;
}
```

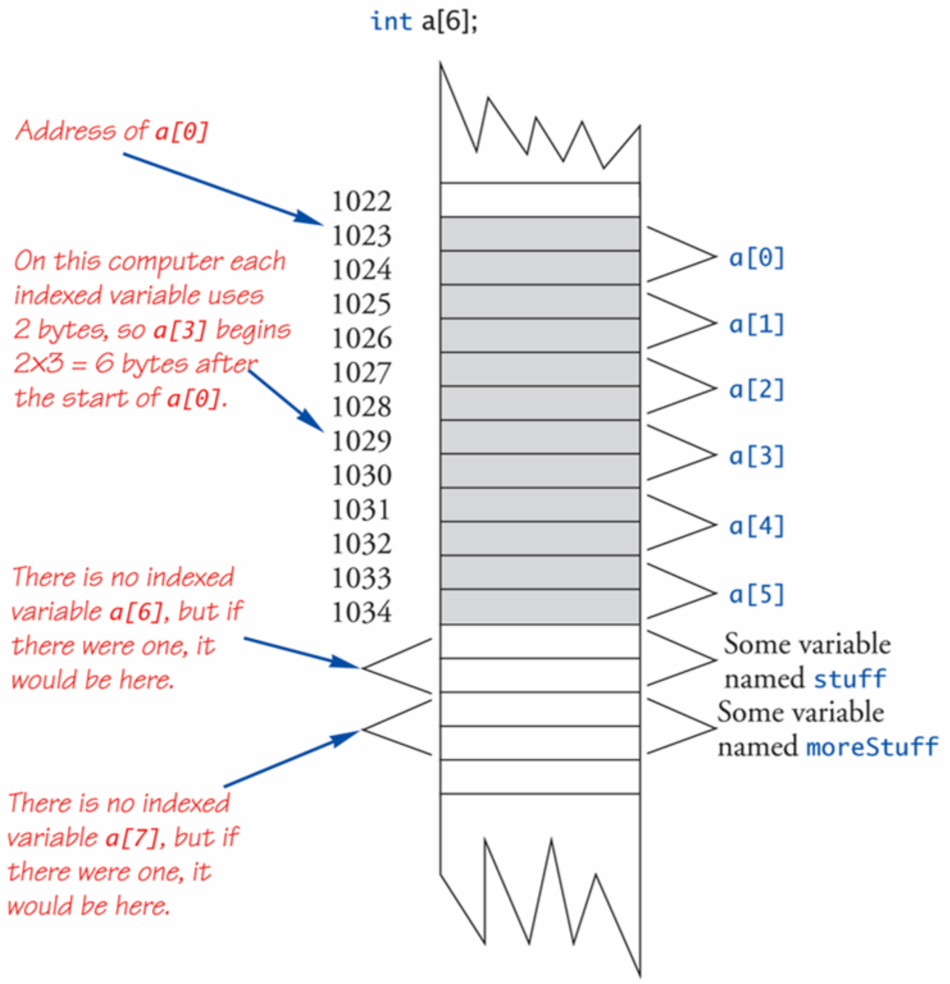
Ambiguous

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"

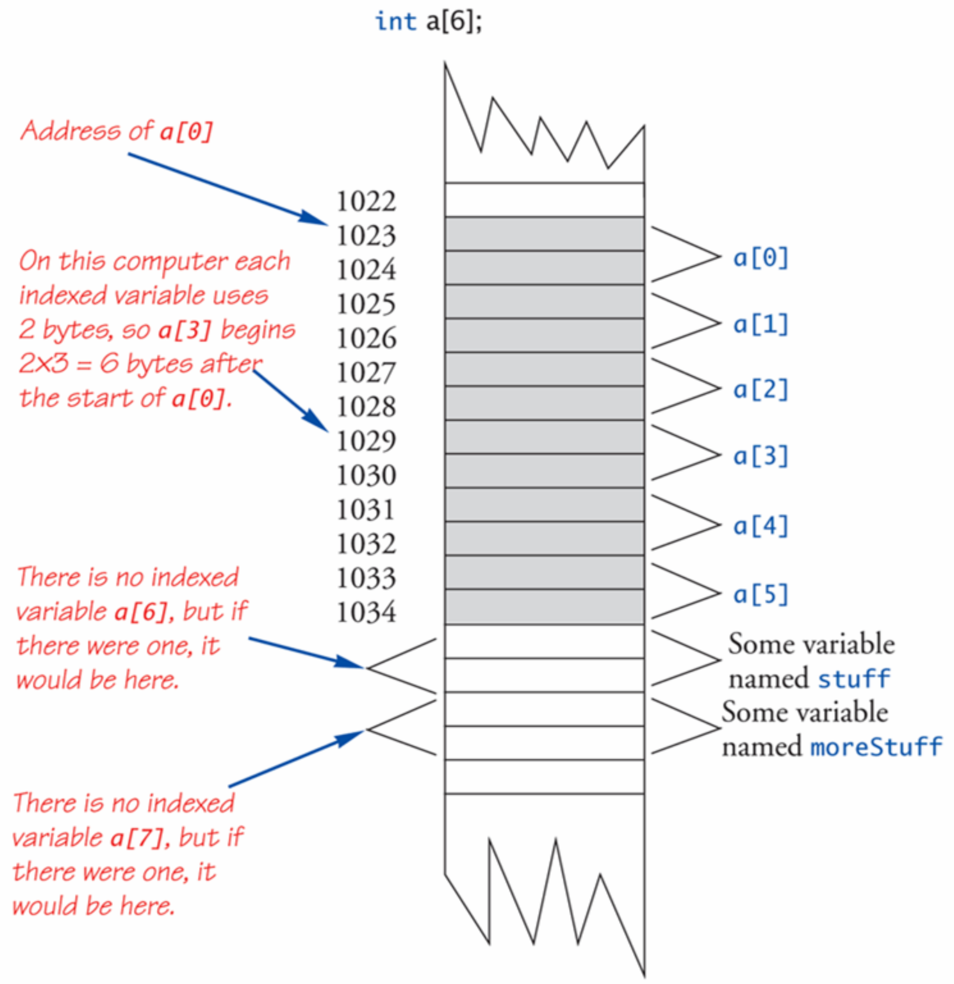
Array declarations allocate memory for entire array

- 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

Arrays

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

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)

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
```

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];
```

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];
```

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.

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 (as Arguments in Functions)

- Entire arrays (passed by the array's name)
  Example code inside a program `main()`:

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

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

Brackets in function definition.

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

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]
```

Array of Arrays

- C++ allows any number of indexes

  Typically no more than two or three

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 !