Introduction to C++

with content from www.cplusplus.com

1

Introduction

- C++
  - widely-used general-purpose programming language
    - compiled
    - procedural and object-oriented support
    - strong library support
  - created by Bjarne Stroustrup starting in 1979
  - based on C
    - first called “C with Classes”
    - also with inheritance, inlining, default function arguments, and strong type checking
    - many C programs compile with C++ compiler
  - major releases in 1983, 1989, 1998, 2011 (C++11)

2

2

Structure of a C++ Program

```
1 // my first program in C++
2 #include <iostream>
3
4 int main()
5 {
6   std::cout << "Hello World!";
7 }
```

```
Hello World!
```

3

3

Structure of a C++ Program

- previous program could also be written as follows

```
int main () { std::cout << " Hello World! "; std::cout << " I'm a C++ program "; }
```

```
1 int main ()
2 {
3   std::cout <<
4     "Hello World!";
5   std::cout
6     << "I'm a C++ program";
7 }
```

4

4

Structure of a C++ Program

- two styles of comments

```
1  /* my second program in C++
2     with more comments */
3
4  #include <iostream>
5
6  int main ()
7  {
8    std::cout << "Hello World! ";     // prints Hello World!
9    std::cout << "I'm a C++ program"; // prints I'm a C++ program
10 }
```

5

5

Structure of a C++ Program

- namespace

```
1 // my second program in C++
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7   cout << "Hello World! ";
8   cout << "I'm a C++ program";
9 }
```

6

6

Identifiers

- similar to rules for Python identifiers
- case-sensitive
- keywords

```
alignas, alignof, and, and_eq, asm, auto, bitand, bitor, bool, break, case, catch, char, char16_t, char32_t,
class, compl, const, constexpr, const_cast, continue, decltype, default, delete, do, double, dynamic_cast,
else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable,
namespace, new, noexcept, not, not_eq, nullptr, operator, or, or_eq, private, protected, public, register,
reinterpret_cast, return, short, signed, sizeof, static, static_assert, static_cast, struct, switch,
template, this, thread_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual,
void, volatile, wchar_t, while, xor, xor_eq
```

7

7

Types

Group	Type names*	Notes on size / precision
Character types	`char`	Exactly one byte in size. At least 8 bits.
	`char16_t`	Not smaller than `char`. At least 16 bits.
	`char32_t`	Not smaller than `char16_t`. At least 32 bits.
	`wchar_t`	Can represent the largest supported character set.
Integer types (signed)	`signed char`	Same size as `char`. At least 8 bits.
	`signed` `short` *`int`*	Not smaller than `char`. At least 16 bits.
	`signed` `int`	Not smaller than `short`. At least 16 bits.
	`signed` `long` *`int`*	Not smaller than `int`. At least 32 bits.
	`signed` `long long` *`int`*	Not smaller than `long`. At least 64 bits.
Integer types (unsigned)	`unsigned char`	(same size as their signed counterparts)
	`unsigned short` *`int`*	
	`unsigned` *`int`*	
	`unsigned long` *`int`*	
	`unsigned long long` *`int`*	
Floating-point types	`float`	
	`double`	Precision not less than `float`
	`long double`	Precision not less than `double`
Boolean type	`bool`	
Void type	`void`	no storage
Null pointer	`decltype(nullptr)`	

8

8

Variables

- must be declared

```
// operating with variables

#include <iostream>
using namespace std;

int main ()
{
  // declaring variables:
  int a, b;
  int result;

  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;

  // print out the result:
  cout << result;

  // terminate the program:
  return 0;
}
```

9

9

Initializing Variables

- different ways to initialize variables at declaration

```
// initialization of variables

#include <iostream>
using namespace std;

int main ()
{
  int a=5;               // initial value: 5
  int b(3);              // initial value: 3
  int c{2};              // initial value: 2
  int result;            // initial value undetermined

  a = a + b;
  result = a - c;
  cout << result;

  return 0;
}
```

10

10

Variables

- automatic type deduction
  - with initialization

```
int foo = 0;
auto bar = foo;  // the same as: int bar = foo;
```

  - without initialization

```
int foo = 0;
decltype(foo) bar;  // the same as: int bar;
```

- used in cases where type cannot be obtained easily for generality

11

11

Strings

```
// my first string
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystring;
  mystring = "This is the initial string content";
  cout << mystring << endl;
  mystring = "This is a different string content";
  cout << mystring << endl;
  return 0;
}
```

12

12

Literals

- integers

```
75          // decimal
0113        // octal
0x4b        // hexadecimal
```

- floats

```
3.14159     // 3.14159
6.02e23     // 6.02 x 10^23
1.6e-19     // 1.6 x 10^-19
3.0         // 3.0
```

- chars

```
'z'
'p'
"Hello world"
"How do you do?"
```

13

13

Literals

- escape sequences

Escape code	Description
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\b	backspace
\f	form feed (page feed)
\a	alert (beep)
\'	single quote (')
\"	double quote (")
\?	question mark (?)
\\	backslash (\)

14

14

Constants

- typed constants

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

const double pi = 3.14159;
const char newline = '\n';

int main ()
{
  double r=5.0;               // radius
  double circle;

  circle = 2 * pi * r;
  cout << circle;
  cout << newline;
}
```

```
31.4159
```

15

15

Constants

- #define constants

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

#define PI 3.14159
#define NEWLINE '\n'

int main ()
{
  double r=5.0;               // radius
  double circle;

  circle = 2 * PI * r;
  cout << circle;
  cout << NEWLINE;

}
```

```
31.4159
```

16

16

Increment/Decrement

- prefix/postfix

Example 1	Example 2

x = 3;
y = ++x;
// x contains 4, y contains 4
	x = 3;
	y = x++;
	// x contains 4, y contains 3

17

17

Operators

- if a=2, b=3, c=6

```
(a == 5)     // evaluates to false, since a is not equal to 5
(a*b >= c)   // evaluates to true, since (2*3 >= 6) is true
(b+4 > a*c)  // evaluates to false, since (3+4 > 2*6) is false
((b=2) == a) // evaluates to true
```

- AND/OR

```
( (5 == 5) && (3 > 6) )  // evaluates to false ( true && false )
( (5 == 5) || (3 > 6) )  // evaluates to true ( true || false )
```

- other operators work similarly to Python

18

18

Ternary Operator

- condition ? result1 : result2

```
7==5 ? 4 : 3     // evaluates to 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3   // evaluates to 4, since 7 is equal to 5+2.
5>3 ? a : b      // evaluates to the value of a, since 5 is greater than 3.
a>b ? a : b      // evaluates to whichever is greater, a or b.
```

```
// conditional operator
#include <iostream>
using namespace std;

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

  a=2;
  b=7;
  c = (a>b) ? a : b;

  cout << c << '\n';
}
```

19

19

Bitwise Operators

operator	asm equivalent	description
&	AND	Bitwise AND
|	OR	Bitwise inclusive OR
^	XOR	Bitwise exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift bits left
>>	SHR	Shift bits right

20

20

Type Casting

- both OK

```
int i;
float f = 3.14;
i = (int) f;
```

```
i = int (f);
```

21

21

Operator Precedence

Level	Precedence group	Operator	Description	Grouping
1	Scope	::	scope qualifier	Left-to-right
2	Postfix (unary)	++ --	postfix increment / decrement	Left-to-right
		()	functional forms	
		[]	subscript	
		. ->	member access	
3	Prefix (unary)	++ --	prefix increment / decrement	Right-to-left
		~ !	bitwise NOT / logical NOT	
		+ -	unary prefix	
		& *	reference / dereference	
		new delete	allocation / deallocation	
		sizeof	parameter pack	
		(type)	C-style type-casting	
4	Pointer-to-member	.* ->*	access pointer	Left-to-right
5	Arithmetic: scaling	* / %	multiply, divide, modulo	Left-to-right
6	Arithmetic: addition	+ -	addition, subtraction	Left-to-right
7	Bitwise shift	<< >>	shift left, shift right	Left-to-right
8	Relational	< > <= >=	comparison operators	Left-to-right
9	Equality	== !=	equality / inequality	Left-to-right
10	And	&	bitwise AND	Left-to-right
11	Exclusive or	^	bitwise XOR	Left-to-right
12	Inclusive or	|	bitwise OR	Left-to-right
13	Conjunction	&&	logical AND	Left-to-right
14	Disjunction	||	logical OR	Left-to-right
15	Assignment-level expressions	= *= /= %= += -= >>= <<= &= ^= |=	assignment / compound assignment	Right-to-left
		?:	conditional operator	
16	Sequencing	,	comma separator	Left-to-right

22

22

Input/Output

```
// i/o example

#include <iostream>
using namespace std;

int main ()
{
  int i;
  cout << "Please enter an integer value: ";
  cin >> i;
  cout << "The value you entered is " << i;
  cout << " and its double is " << i*2 << ".\n";
  return 0;
}
```

```
Please enter an integer value: 702
The value you entered is 702 and its double is 1404.
```

23

23

Input/Output

```
// cin with strings
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystr;
  cout << "What's your name? ";
  getline (cin, mystr);
  cout << "Hello " << mystr << ".\n";
  cout << "What is your favorite team? ";
  getline (cin, mystr);
  cout << "I like " << mystr << " too!\n";
  return 0;
}
```

```
What's your name? Homer Simpson
Hello Homer Simpson.
What is your favorite team? The Isotopes
I like The Isotopes too!
```

24

24

if Statements

- compound if

```
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
```

25

25

Iteration

- while statement

```
// custom countdown using while
#include <iostream>
using namespace std;

int main ()
{
  int n = 10;

  while (n>0) {
    cout << n << ", ";
    --n;
  }

  cout << "liftoff!\n";
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!
```

26

26

Iteration

- do-while statement

```
// echo machine
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str;
  do {
    cout << "Enter text: ";
    getline (cin,str);
    cout << "You entered: " << str << '\n';
  } while (str != "goodbye");
}
```

```
Enter text: hello
You entered: hello
Enter text: who's there?
You entered: who's there?
Enter text: goodbye
You entered: goodbye
```

27

27

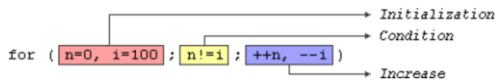
Iteration

- for loop

```
// countdown using a for loop
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "liftoff!\n";
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!
```



28

28

Iteration

- range-based for loop

```
// range-based for loop
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str {"Hello!"};
  for (char c : str)
  {
    std::cout << "[" << c << "]";
  }
  std::cout << '\n';
}
```

```
[H][e][l][l][o][!]
```

29

29

break Statement

- break

```
// break loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown aborted!";
      break;
    }
  }
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!
```

30

30

continue Statement

- continue

```
// continue loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "liftoff!\n";
}
```

```
10, 9, 8, 7, 6, 4, 3, 2, 1, liftoff!
```

31

31

switch Statement

switch example	if-else equivalent

switch (x) {
case 1:
cout << "x is 1";
break;
case 2:
cout << "x is 2";
break;
default:
cout << "value of x unknown";
}
	if (x == 1) {
	cout << "x is 1";
	}
	else if (x == 2) {
	cout << "x is 2";
	}
	else {
	cout << "value of x unknown";
	}

```
switch (x) {
  case 1:
  case 2:
  case 3:
    cout << "x is 1, 2 or 3";
    break;
  default:
    cout << "x is not 1, 2 nor 3";
  }
```

32

32

Functions

```
// function example
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
}
```

```
The result is 8
```

33

33

Functions

```
// function example
#include <iostream>
using namespace std;

int subtraction (int a, int b)
{
  int r;
  r=a-b;
  return r;
}

int main ()
{
  int x=5, y=3, z;
  z = subtraction (7,2);
  cout << "The first result is " << z << '\n';
  cout << "The second result is " << subtraction (7,2) << '\n';
  cout << "The third result is " << subtraction (x,y) << '\n';
  z= 4 + subtraction (x,y);
  cout << "The fourth result is " << z << '\n';
}
```

```
The first result is 5
The second result is 5
The third result is 2
The fourth result is 6
```

34

34

Functions

```
// void function example
#include <iostream>
using namespace std;

void printmessage ()
{
  cout << "I'm a function!";
}

int main ()
{
  printmessage ();
}
```

```
I'm a function!
```

35

35

Functions

- return value from main

value	description
0	The program was successful
EXIT_SUCCESS	The program was successful (same as above). This value is defined in header <cstdlib>.
EXIT_FAILURE	The program failed. This value is defined in header <cstdlib>.

36

36

Functions

- pass by value vs. pass by reference

```
// passing parameters by reference
#include <iostream>
using namespace std;

void duplicate (int& a, int& b, int& c)
{
  a*=2;
  b*=2;
  c*=2;
}

int main ()
{
  int x=1, y=3, z=7;
  duplicate (x, y, z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
  return 0;
}
```

```
x=2, y=6, z=14
```

37

37

Functions

- inline functions

```
inline string concatenate (const string& a, const string& b)
{
  return a+b;
}
```

38

38

Functions

- default values for parameters

```
// default values in functions
#include <iostream>
using namespace std;

int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}

int main ()
{
  cout << divide (12) << '\n';
  cout << divide (20,4) << '\n';
  return 0;
}
```

```
6
5
```

39

39

Functions

- function prototypes

```
// declaring functions prototypes
#include <iostream>
using namespace std;

void odd (int x);
void even (int x);

int main()
{
  int i;
  do {
    cout << "Please, enter number (0 to exit): ";
    cin >> i;
    odd (i);
  } while (i!=0);
  return 0;
}

void odd (int x)
{
  if ((x%2)!=0) cout << "It is odd.\n";
  else even (x);
}

void even (int x)
{
  if ((x%2)==0) cout << "It is even.\n";
  else odd (x);
}
```

```
Please, enter number (0 to exit): 9
It is odd.
Please, enter number (0 to exit): 6
It is even.
Please, enter number (0 to exit): 1030
It is even.
Please, enter number (0 to exit): 0
It is even.
```

40

40

Functions

- recursion

```
// factorial calculator
#include <iostream>
using namespace std;

long factorial (long a)
{
  if (a > 1)
   return (a * factorial (a-1));
  else
   return 1;
}

int main ()
{
  long number = 9;
  cout << number << "! = " << factorial (number);
  return 0;
}
```

```
9! = 362880
```

41

41

Scope

- global vs. local variables

```
int foo;         // global variable

int some_function ()
{
  int bar;       // local variable
  bar = 0;
}

int other_function ()
{
  foo = 1;  // ok: foo is a global variable
  bar = 2;  // wrong: bar is not visible from this function
}
```

- general rule: DO NOT USE

42

42

Scope

- name can only represent one entity

```
int some_function ()
{
  int x;
  x = 0;
  double x;   // wrong: name already used in this scope
  x = 0.0;
}
```

43

43

Scope

- block scope

```
// inner block scopes
#include <iostream>
using namespace std;

int main () {
  int x = 10;
  int y = 20;
  {
    int x;   // ok, inner scope.
    x = 50;  // sets value to inner x
    y = 50;  // sets value to (outer) y
    cout << "inner block:\n";
    cout << "x: " << x << '\n';
    cout << "y: " << y << '\n';
  }
  cout << "outer block:\n";
  cout << "x: " << x << '\n';
  cout << "y: " << y << '\n';
  return 0;
}
```

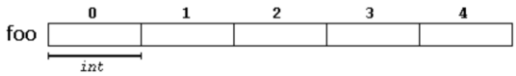
```
inner block:
x: 50
y: 50
outer block:
x: 10
y: 50
```

44

44

Arrays

- contiguous memory locations




```
int foo [5];
```

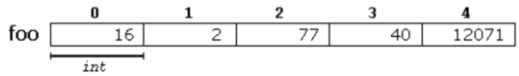
45

45

Initializing Arrays

- elements not automatically initialized, but can be explicitly initialized

```
int foo [5] = { 16, 2, 77, 40, 12071 };
```



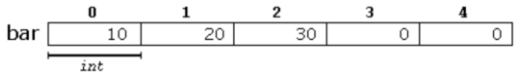
46

46

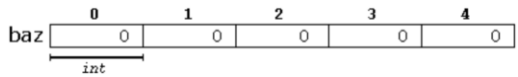
Initializing Arrays

- if { } are present, values are initialized to default values

```
int bar [5] = { 10, 20, 30 };
```




```
int baz [5] = { };
```



47

47

Initializing Arrays

- initialized arrays without size are automatically sized to accommodate values

```
int foo [] = { 16, 2, 77, 40, 12071 };
```

- can be initialized without =

```
int foo[] = { 10, 20, 30 };
int foo[] { 10, 20, 30 };
```

- no error if range exceeded
- example uses of arrays

```
foo[0] = a;
foo[a] = 75;
b = foo [a+2];
foo[foo[a]] = foo[2] + 5;
```

48

48

Arrays

```
// arrays example
#include <iostream>
using namespace std;

int foo [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
  for ( n=0 ; n<5 ; ++n )
  {
    result += foo[n];
  }
  cout << result;
  return 0;
}
```

```
12206
```

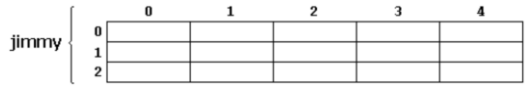
49

49

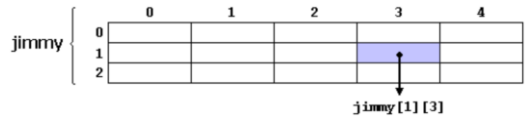
Multidimensional Arrays

- arrays of arrays

```
int jimmy [3][5];
```




```
jimmy[1][3]
```



50

50

Multidimensional Arrays

- can be any dimension, but space increases exponentially

```
char century [100][365][24][60][60];
```

- allocates a char for each second in the last century
- consumes 3GB of memory

- could have been implemented as a single-dimension array

```
int jimmy [3][5];   // is equivalent to
int jimmy [15];     // (3 * 5 = 15)
```

51

51

Multidimensional Arrays

multidimensional array	pseudo-multidimensional array

`#define WIDTH 5`
`#define HEIGHT 3`
 	
`int jimmy [HEIGHT][WIDTH];`
`int n,m;`
 	
`int main ()`
`{`
`  for (n=0; n<HEIGHT; n++)`
`    for (m=0; m<WIDTH; m++)`
`    {`
`      jimmy[n][m]=(n+1)*(m+1);`
`    }`
`}`
	`#define WIDTH 5`
	`#define HEIGHT 3`
	
	`int jimmy [HEIGHT * WIDTH];`
	`int n,m;`
	
	`int main ()`
	`{`
	`  for (n=0; n<HEIGHT; n++)`
	`    for (m=0; m<WIDTH; m++)`
	`    {`
	`      jimmy[n*WIDTH+m]=(n+1)*(m+1);`
	`    }`
	`}`

jimmy

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

52

52

Arrays as Parameters

```
// arrays as parameters
#include <iostream>
using namespace std;

void printarray (int arg[], int length) {
  for (int n=0; n<length; ++n)
    cout << arg[n] << ' ';
  cout << '\n';
}

int main ()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8, 10};
  printarray (firstarray,3);
  printarray (secondarray,5);
}
```

```
5 10 15
2 4 6 8 10
```

```
void procedure (int myarray[][3][4])
```

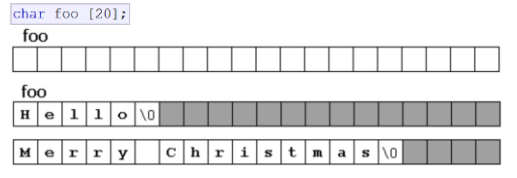
53

53

Character Arrays

- sets aside space, but not initialized

```
char foo [20];
```



54

54

Character Arrays

- can initialize with individual elements or string literals

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char myword[] = "Hello";
```

- not valid

```
myword = "Bye";
myword[] = "Bye";
```

```
myword = { 'B', 'y', 'e', '\0' };
```

- OK

```
myword[0] = 'B';
myword[1] = 'y';
myword[2] = 'e';
myword[3] = '\0';
```

55

55

Strings and Character Arrays

```
// strings and NTCS:
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  char question1[] = "What is your name? ";
  string question2 = "Where do you live? ";
  char answer1 [80];
  string answer2;
  cout << question1;
  cin >> answer1;
  cout << question2;
  cin >> answer2;
  cout << "Hello, " << answer1;
  cout << " from " << answer2 << "!\n";
  return 0;
}
```

```
What is your name? Homer
Where do you live? Greece
Hello, Homer from Greece!
```

56

56

Strings and Character Arrays

- can be transformed one to another

```
char myntcs[] = "some text";
string mystring = myntcs;  // convert c-string to string
cout << mystring;          // printed as a library string
cout << mystring.c_str();  // printed as a c-string
```

57

57

Pointers

- pointer – the address of something
- exact memory locations unknown at compile time

- use & to get the address of a variable

```
foo = &myvar;
```

```
myvar = 25;
foo = &myvar;
bar = myvar;
```

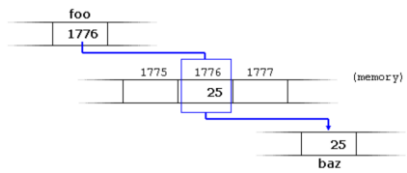
58

58

Pointers

- use * to get the value at a pointer (address)

```
baz = *foo;
```



59

59

Pointers

- & and * are complementary

```
baz = foo;   // baz equal to foo (1776)
baz = *foo;  // baz equal to value pointed to by foo (25)
```

- with following assignments

```
myvar = 25;
foo = &myvar;
```

- all of the following are true

```
myvar == 25
&myvar == 1776
foo == 1776
*foo == 25
```

```
*foo == myvar
```

60

60

Declaring Pointers

- all are the same size in memory

```
int * number;
char * character;
double * decimals;
```

- different

```
int * p1, * p2;
```

```
int * p1, p2;
```

61

61

Pointers

```
// my first pointer
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;
  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

62

62

Pointers

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed to by p1 = 10
  *p2 = *p1;         // value pointed to by p2 = value pointed to by p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed to by p1 = 20

  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

63

63

Pointers and Arrays

- array name with no index is a pointer to the first element
- arrays can always be converted to pointers

```
int myarray [20];
int * mypointer;
```

```
mypointer = myarray;
```

- not valid to go the other way

```
myarray = mypointer;
```

64

64

Pointers and Arrays

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;
}
```

```
10, 20, 30, 40, 50,
```

65

65

Pointers and Arrays

- array with index is a simply a pointer with an offset
- can be represented with pointer

```
a[5] = 0;       // a [offset of 5] = 0
*(a+5) = 0;     // pointed to by (a+5) = 0
```

66

66

Pointer Initialization

- pointers can be initialized at declaration

```
int myvar;
int * myptr = &myvar;
```

- same as

```
int myvar;
int * myptr;
myptr = &myvar;
```

- not valid

```
int myvar;
int * myptr;
*myptr = &myvar;
```

- OK

```
int myvar;
int *foo = &myvar;
int *bar = foo;
```

67

67

Pointer Arithmetic

- pointers can be used in arithmetic expressions, with underlying size taken into account
- suppose the following have addresses 1000, 2000, 3000

```
char *mychar;
short *myshort;
long *mylong;
```

- after the following

```
++mychar;
++myshort;
++mylong;
```

  - values are 1001, 2002, 3004
- same results for

```
mychar = mychar + 1;
myshort = myshort + 1;
mylong = mylong + 1;
```

68

68

Pointer Arithmetic

- the following is equivalent to *(p++)

```
*p++
```

- other examples

```
*p++   // same as *(p++): increment pointer, and dereference unincremented address
*++p   // same as *(++p): increment pointer, and dereference incremented address
++*p   // same as ++(*p): dereference pointer, and increment the value it points to
(*p)++ // dereference pointer, and post-increment the value it points to
```

- assignment done before increment

```
*p++ = *q++;
```

- same as

```
*p = *q;
++p;
++q;
```

69

69

Pointers and const

- if value pointed to is const, it cannot be modified

```
int x;
int y = 10;
const int * p = &y;
x = *p;          // ok: reading p
*p = x;          // error: modifying p, which is const-qualified
```

- pointers can be const

```
int x;
      int *       p1 = &x;  // non-const pointer to non-const int
const int *       p2 = &x;  // non-const pointer to const int
      int * const p3 = &x;  // const pointer to non-const int
const int * const p4 = &x;  // const pointer to const int
```

- same

```
const int * p2a = &x;  //       non-const pointer to const int
int const * p2b = &x;  // also non-const pointer to const int
```

70

70

Pointers and const

- pointers are not const, so can be modified

```
// pointers as arguments:
#include <iostream>
using namespace std;

void increment_all (int* start, int* stop)
{
  int * current = start;
  while (current != stop) {
    ++(*current);  // increment value pointed
    ++current;     // increment pointer
  }
}

void print_all (const int* start, const int* stop)
{
  const int * current = start;
  while (current != stop) {
    cout << *current << '\n';
    ++current;     // increment pointer
  }
}

int main ()
{
  int numbers[] = {10,20,30};
  increment_all (numbers,numbers+3);
  print_all (numbers,numbers+3);
  return 0;
}
```

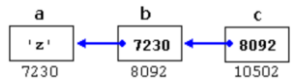
```
11
21
31
```

71

71

Pointers to Pointers

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```



72

72

void Pointers

- void pointers point to no particular type

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

void increase (void* data, int psize)
{
  if ( psize == sizeof(char) )
  { char* pchar; pchar=(char*)data; ++(*pchar); }
  else if (psize == sizeof(int) )
  { int* pint; pint=(int*)data; ++(*pint); }
}

int main ()
{
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << '\n';
  return 0;
}
```

```
y, 1603
```

73

73

Pointers

- pointers can point to any address

```
int * p;               // uninitialized pointer (local variable)

int myarray[10];
int * q = myarray+20;  // element out of bounds
```

- pointers can point to nothing

```
int * p = 0;
int * q = nullptr;
```

- or simply

```
int * r = NULL;
```

- NULL pointers and void pointers are different

74

74

Dynamic Memory

- memory can be allocated during run time with new

```
int * foo;
foo = new int [5];
```

- can check for success/failure

```
int * foo;
foo = new (nothrow) int [5];
if (foo == nullptr) {
  // error assigning memory. Take measures.
}
```

75

75

Dynamic Memory

- memory can (and should) be de-allocated during run time with delete

```
delete pointer;
delete[] pointer;
```

- can also use malloc/free (from C), but don't mix

76

76

Dynamic Memory

```
// rememb-o-matic
#include <iostream>
#include <new>
using namespace std;

int main ()
{
  int i,n;
  int * p;
  cout << "How many numbers would you like to type? ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == nullptr)
    cout << "Error: memory could not be allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
  }
  return 0;
}
```

```
How many numbers would you like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,
```

77

77

Data Structures

- struct

```
struct product {
  int weight;
  double price;
} ;

product apple;
product banana, melon;
```

- or

```
struct product {
  int weight;
  double price;
} apple, banana, melon;
```

- access

```
apple.weight
apple.price
banana.weight
banana.price
melon.weight
melon.price
```

78

78

Data Structures

```
// example about structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
} mine, yours;

void printmovie (movies_t movie);

int main ()
{
  string mystr;

  mine.title = "2001 A Space Odyssey";
  mine.year = 1968;

  cout << "Enter title: ";
  getline (cin,yours.title);
  cout << "Enter year: ";
  getline (cin,mystr);
  stringstream(mystr) >> yours.year;

  cout << "My favorite movie is:\n ";
  printmovie (mine);
  cout << "And yours is:\n ";
  printmovie (yours);
  return 0;
}

void printmovie (movies_t movie)
{
  cout << movie.title;
  cout << " (" << movie.year << ")\n";
}
```

```
Enter title: Alien
Enter year: 1979

My favorite movie is:
 2001 A Space Odyssey (1968)
And yours is:
 Alien (1979)
```

79

79

Data Structures

```
// array of structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
} films [3];

void printmovie (movies_t movie);

int main ()
{
  string mystr;
  int n;

  for (n=0; n<3; n++)
  {
    cout << "Enter title: ";
    getline (cin,films[n].title);
    cout << "Enter year: ";
    getline (cin,mystr);
    stringstream(mystr) >> films[n].year;
  }

  cout << "\nYou have entered these movies:\n";
  for (n=0; n<3; n++)
    printmovie (films[n]);
  return 0;
}

void printmovie (movies_t movie)
{
  cout << movie.title;
  cout << " (" << movie.year << ")\n";
}
```

```
Enter title: Blade Runner
Enter year: 1982
Enter title: The Matrix
Enter year: 1999
Enter title: Taxi Driver
Enter year: 1976

You have entered these movies:
Blade Runner (1982)
The Matrix (1999)
Taxi Driver (1976)
```

80

80

Data Structures

```
// pointers to structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
};

int main ()
{
  string mystr;

  movies_t amovie;
  movies_t * pmovie;
  pmovie = &amovie;

  cout << "Enter title: ";
  getline (cin, pmovie->title);
  cout << "Enter year: ";
  getline (cin, mystr);
  (stringstream) mystr >> pmovie->year;

  cout << "\nYou have entered:\n";
  cout << pmovie->title;
  cout << " (" << pmovie->year << ")\n";

  return 0;
}
```

```
Enter title: Invasion of the body snatchers
Enter year: 1978

You have entered:
Invasion of the body snatchers (1978)
```

81

81

Data Structures

- pointers to structs

```
pmovie->title
```

```
(*pmovie).title
```

- different from

```
*pmovie.title
```

```
*(pmovie.title)
```

Expression	What is evaluated	Equivalent
a.b	Member b of object a	
a->b	Member b of object pointed to by a	(*a).b
*a.b	Value pointed to by member b of object a	*(a.b)

82

82

Data Structures

- nested structs

```
struct movies_t {
  string title;
  int year;
};

struct friends_t {
  string name;
  string email;
  movies_t favorite_movie;
} charlie, maria;

friends_t * pfriends = &charlie;
```

- access

```
charlie.name
maria.favorite_movie.title
charlie.favorite_movie.year
pfriends->favorite_movie.year
```

83

83

Other Data Structures

- type aliases

```
typedef char C;
typedef unsigned int WORD;
typedef char * pChar;
typedef char field [50];
```

- can be used as

```
C mychar, anotherchar, *ptc1;
WORD myword;
pChar ptc2;
field name;
```

- with using clause

```
using C = char;
using WORD = unsigned int;
using pChar = char *;
using field = char [50];
```

84

84

Other Data Structures

- union

```
union mytypes_t {
  char c;
  int i;
  float f;
} mytypes;
```

- can be accessed as

```
mytypes.c
mytypes.i
mytypes.f
```

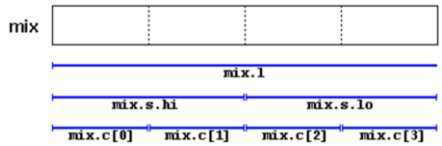
85

85

Other Data Structures

- union

```
union mix_t {
  int l;
  struct {
    short hi;
    short lo;
    } s;
  char c[4];
} mix;
```



86

86

Other Data Structures

- anonymous union

structure with regular union	structure with anonymous union

struct book1_t {
  char title[50];
  char author[50];
  union {
  float dollars;
  int yen;
  } price;
} book1;
	struct book2_t {
	  char title[50];
	  char author[50];
	  union {
	  float dollars;
	  int yen;
	  };
	} book2;

```
book1.price.dollars
book1.price.yen
```

```
book2.dollars
book2.yen
```

87

87

Enumerated Types

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

```
colors_t mycolor;

mycolor = blue;
if (mycolor == green) mycolor = red;
```

- can assign integer values (assigned anyway starting at 0)

```
enum months_t { january=1, february, march, april,
                may, june, july, august,
                september, october, november, december} y2k;
```

88

88