C++

short reference

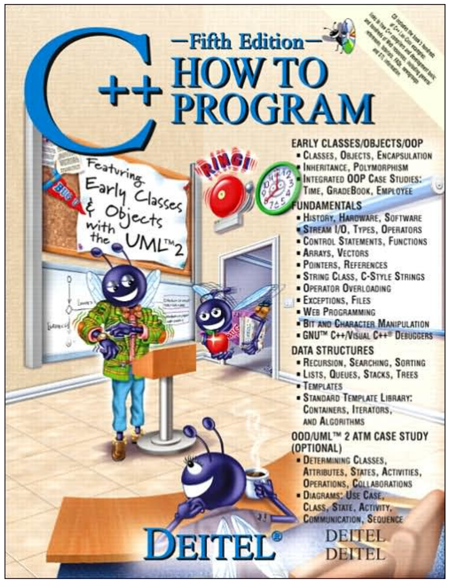
for

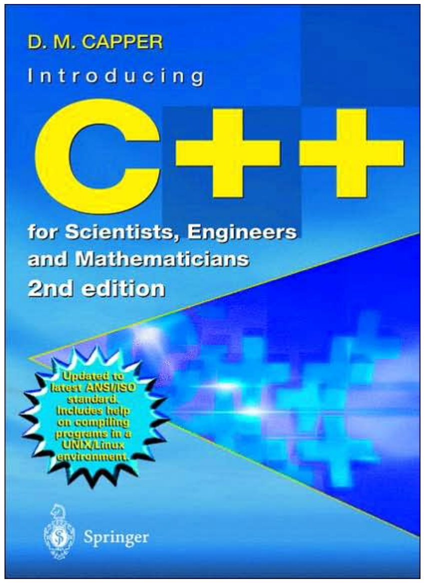
introductory computational physics

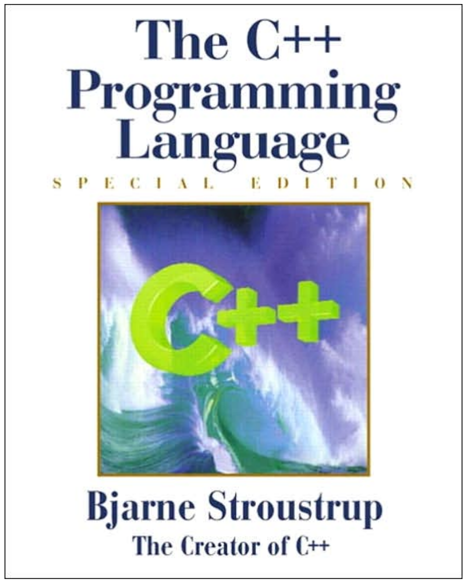
Short introduction to C++

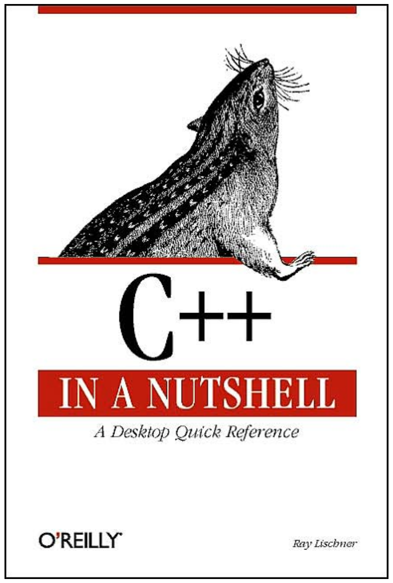
- Structure of a program
- Variables, Data Types, and Constants
- Operators
- Basic Input/Output
- Control Structures
- Functions
- Arrays
- Input/Output with files
- Pointers
- Classes

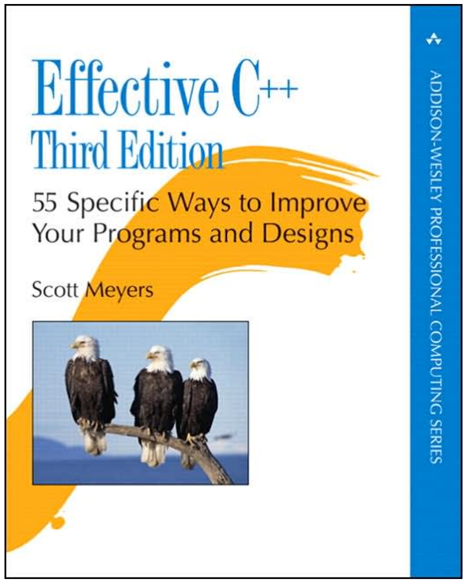
Reference books

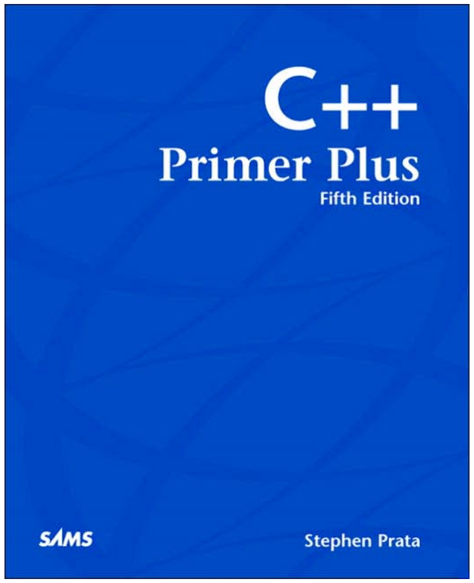


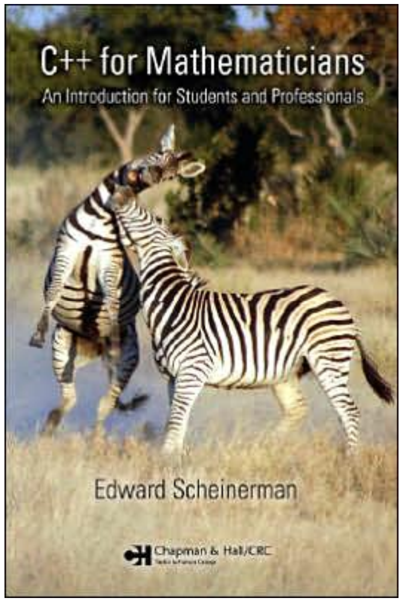







Effective C++
Third Edition
55 Specific Ways to Improve Your Programs and Designs
Scott Meyers
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES





… and many more!!!

Good practice

Have a good reference book for the version of C+ + you are using.

Refer to this book frequently to be sure you are aware of the rich collection of C++ features and you are using these features correctly.

Programming tips

Some books* have very practical advice on

# Good programming practices

# Common programming errors

# Performance tips

# Software engineering observations

# Testing and debugging tips

* *C++ how to program*, Deitel & Deitel have hundreds of valuable tips.

Part 1

Structure of a program

```
// Simple program
#include <iostream>
using namespace std;
int main()
{
    int x, y;

    x = 2;
    y = x + 4;
    cout <<" x = "<<x<<"  x + 4 = "<<y << endl;

    return 0;
}
```

```
x = 2  x + 2 = 4
```

statements

more complex structure involves programmer-defined functions, control statements, classes, communication with files, …

Free-format language

C++ is a free-format language like many other languages.

The compiler ignores ALL spaces, tabs, and new-line characters (also called “white spaces”)

The compiler recognizes “white spaces” only inside a string.

Using white spaces allows to better visualize a program structure (e.g. extra indentation inside **if** statements, **for** loops, etc.) .

Common structure of a program

1. Comments
2. Header files
3. Declare variables
4. Declare constants
5. Read initial data
6. Open files
7. CALCULATIONS (include calling other functions)
8. Write results
9. Closing
10. Stop

Steps 5 – 9 may call other modules

Part 2

Variables, Data Types, and Constants

Variables, Data Types and Constants

- Identifiers (names of variables)
- Fundamental data types
- Declaration of variables
- Global and local variables
- Initialization of variables
- Constants

Variables

Variable is a location in the computer’s memory where a value can be stored for use by a program.

Identifiers - Names of variables

A variable name is any valid *identifier*.

An identifier is a series of characters consisting of letters, digits, and uderscore (_) that does not begin with a digit.

C++ is *case sensitive* – uppercase and lowercase letters are different.

Examples:

```
abc
velocity_i
Force_12
```

Identifiers: reserved key words

These keywords must not be used as identifiers!

C and C++ keywords

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

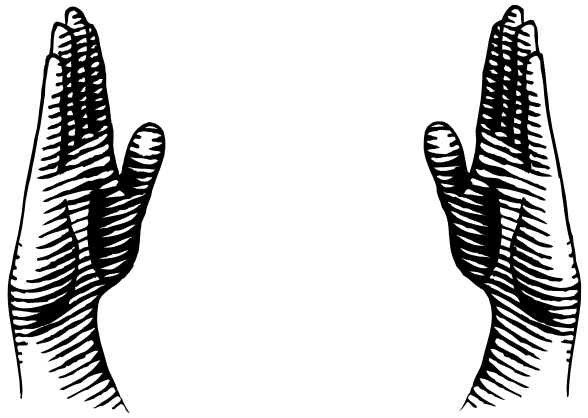
Identifiers: reserved key words II

C++ only keywords

asm	bool	catch	class	const_cast
delete	dynamic_cast	explicit	false	friend
inline	mutable	namespace	new	operator
private	protected	public	reinterpret_cast	
static_cast	template	this	throw	true
try	typeid	typename	using	virtual
wchar_t				

Variables: Data Types

Each variable has a name, a type, a size and a value.



Fundamental data types in C++

name	description	bytes
char	Character or small integer	1
short int	Short Integer	2
int	Integer	4
long int	Long integer	4*
bool	Boolean	1
float	Floating point number	4
double	Double precision floating point	8
long double	Long double precision	8*
wchar_t	Wide character	2

* depends on a system

Range of data types in C++

name	range	bytes
short int	signed: -32768 to 32767 unsigned: 0 to 65535	2
int	-2147483648 to 2147483647 unsigned: 0 to 4294967295	4
bool	true or false	1
float	3.4e +/- 38 (7 digits)	4
double	1.7e +/- 308 (15 digits)	8
long double	1.7e +/- 308 (15 digits)	8*

Declaration of variables

All variables must be declared with a name and a data type before they can be used by a program.

```
//declaration of variables
#include <iostream>
using namespace std;
int main()
{
    double a, speed, force_12;
    int   i, n;
    ... some operators ...
    return 0;
}
```

Global and local variables

A global variable is a variable declared in the main body of the source code, outside all functions.

Global variables can be referred from anywhere in the code, even inside functions,

A local variable is one declared within the body of a function or a block.

The scope of local variables is limited to the block enclosed in braces {} where they are declared.

```cpp
// test on global and local variables
#include <iostream>
using namespace std;
void f12(void);
int nglobal = 1;
int main()
{
  cout << "main 1: nglobal = " << nglobal <<endl;
  nglobal = 2;
  cout << "main 2: nglobal = " << nglobal <<endl;
  f12();
  cout << "main 3: nglobal = " << nglobal <<endl;

}
void f12()
{
  cout << "f12    : nglobal = " << nglobal <<endl;
  nglobal = 3;
}
```

```
main 1: nglobal = 1
main 2: nglobal = 2
f12   : nglobal = 2
main 3: nglobal = 3
```

Initialization of variables

When declaring a regular local variable, its value is by default undetermined.

Initialization 1:

type identifier = initial_value;

```
float sum = 0.0;
```

Initialization 2:

type identifier (initial_value) ;

```
float sum (0.0);
```

Constants

Declared constants

const type identifier = initial_value ;

Constant variable can not be modified thereafter.

```
const double pi = 3.1415926;
```

Define constants

#define identifier value

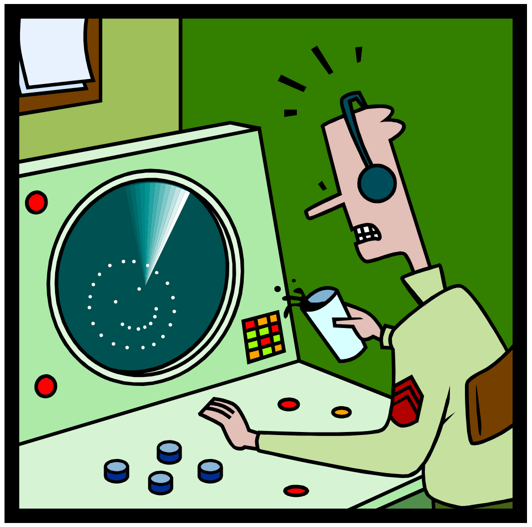
```
#define PI 3.14159265
```

Example

```
//declaration of variables (example)
#include <iostream>
using namespace std;
#define PI 3.1415926
const float Ry = 13.6058;
int main()
{
    float a, speed, force_12;
    int   i, n;
    float angle = 45.0;
    ... some operators ...
    return 0;
}
```

Part 3

Operators



Operators

- Assignment (=)
- Arithmetic operators (+, -, *, /, %)
- Compound assignation (+=, -=, *=, /=, %=)
- Increment and decrement (++, --)
- Relational and equality operators (==, !=, >, <, >=, <=)
- Logical operators (!, &&, ||)
- Conditional operator (?)
- Comma operator (,)
- Precedence of operators

Assignment operator (=)

The assignment operator assigns a value to a variable.

```
// operator (=)
#include <iostream>
using namespace std;
int main ()
{
   int a, b;
   a = 12;
   b = a;
   cout << " a = " << a
        << " b = " << b <<endl;
   return 0;
}
```

```
a = 12 b = 12
```

Arithmetic operators

There are five arithmetic operators

Operator	Symbol	C++ example
addition	+	f + 7
subtraction	-	p - c
multiplication	*	b * k
division	/	x / y
modulus	%	r % s

Precedence of arithmetic operators

Operator	Operation	Order
()	Parentheses	Evaluated first
*, / ,%	Multiplication	Evaluated second
	Division	(if more than one
	Modulus	then left-to-right)
+, -	Addition	Evaluated last
	Subtraction	(if more than
one		
		then left-to-right)

Arithmetic assignment operators

There are five arithmetic assignment operators

Operator	C++	explanation
+=	a += 7	a = a + 7
-=	b -= 4	b = b - 4
*=	c *= 5	c = c * 5
/=	d /= 3	d = d / 3
%=	e %= 9	e = e % 9

The increment/decrement operators

Operator	called	C++
++	pre increment	++a
++	post increment	a++
--	pre decrement	--a
--	post decrement	a--

Equality and relational operators

Equality operators in decision making

	C++	example	meaning
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y

Relational operators in decision making

	C++	example	meaning
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater or equal to y
≤	<=	x <= y	x is less than or equal to y

Logical operators

C++ provides *logical operators* that are used to form complex conditions by combining simple conditions.

Operator	Symbols	C++ example
and	&&	if (i==1 && j>=10)
or	||	if (speed >= 10.0 || t <=2.0)

Condition operator ?

The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format

condition ? result1 : result2

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

int main ()
{
int a,b,c;
   a=2;
   b=7;
   c = (a>b) ? a : b;
   cout << " c = " << c;
   return 0;
}
```

```
c = 7
```

Part 4

Basic Input/Output

Input/Output

The C++ libraries provide an extensive set of input/output capabilities.

C++ I/O occurs in *stream of bytes*.

Iostream Library header files

<iostream.h>	contains cin, cout, cerr, clog.
<iomanip.h>	information for formatting
<fstream.h>	for file processing

Basic Input/Output

cin is an object of the istream class and is connected to the standard input device (normally the keyboard)

cout is an object of the ostream class and is connected to the standard output device (normally the screen)

Example (output)

```
// output
#include <iostream.h>
int main ()
{
int a;
   a=2;
   cout << " a = " << a << endl;
return 0;
}
```

```
a = 2
```

Example (input/output)

```
// Input and output
#include <iostream.h>
int main ()
{
int a, b;
   cout << " enter two integers:";
   cin >> a >> b;
   cout << " a = " << a
        << " b = " << b << endl;
return 0;
}
```

```
enter two integers:2 4
a = 2 b = 4
```

Elements of formatting

setw set the field width (positions for input/output)

setprecision control the precision of float-point numbers

setiosflags(ios::fixed | ios::showpoint) sets fixed point output with a decimal point

```
cout << setw(5)<< n
<< setw(10)<< setprecision(4)
<< setiosflags(ios::fixed | ios::showpoint)
<< t <<endl;
```

for n = 2 and t = 4.0 $\longrightarrow$

```
    2    4.0000
```

Some format state flags

ios :: showpoint Specify that floating-point numbers should be output with a decimal

ios::fixed Specify output of a floating-point value in fixed-point notation with a specific number of digits to the right of the decimal point.

ios::scientific Specify output of a floating-point value in scientific notation.

ios::left Left justify output in a field.

ios::right Right justify output in a field.

Example

```
  cout.setf(ios::fixed | ios::showpoint);
  cout.width(10);
  cout.precision(5);
cout << "radius   = " << radius   << endl;
cout << "diameter = " << diameter<< endl;
cout << "circumf. = " << circumf << endl;
cout << "area     = " << area     << endl;
```

```
radius   = 3.00000
diameter = 6.00000
circumf. = 18.84956
area     = 28.27433
```

Part 5

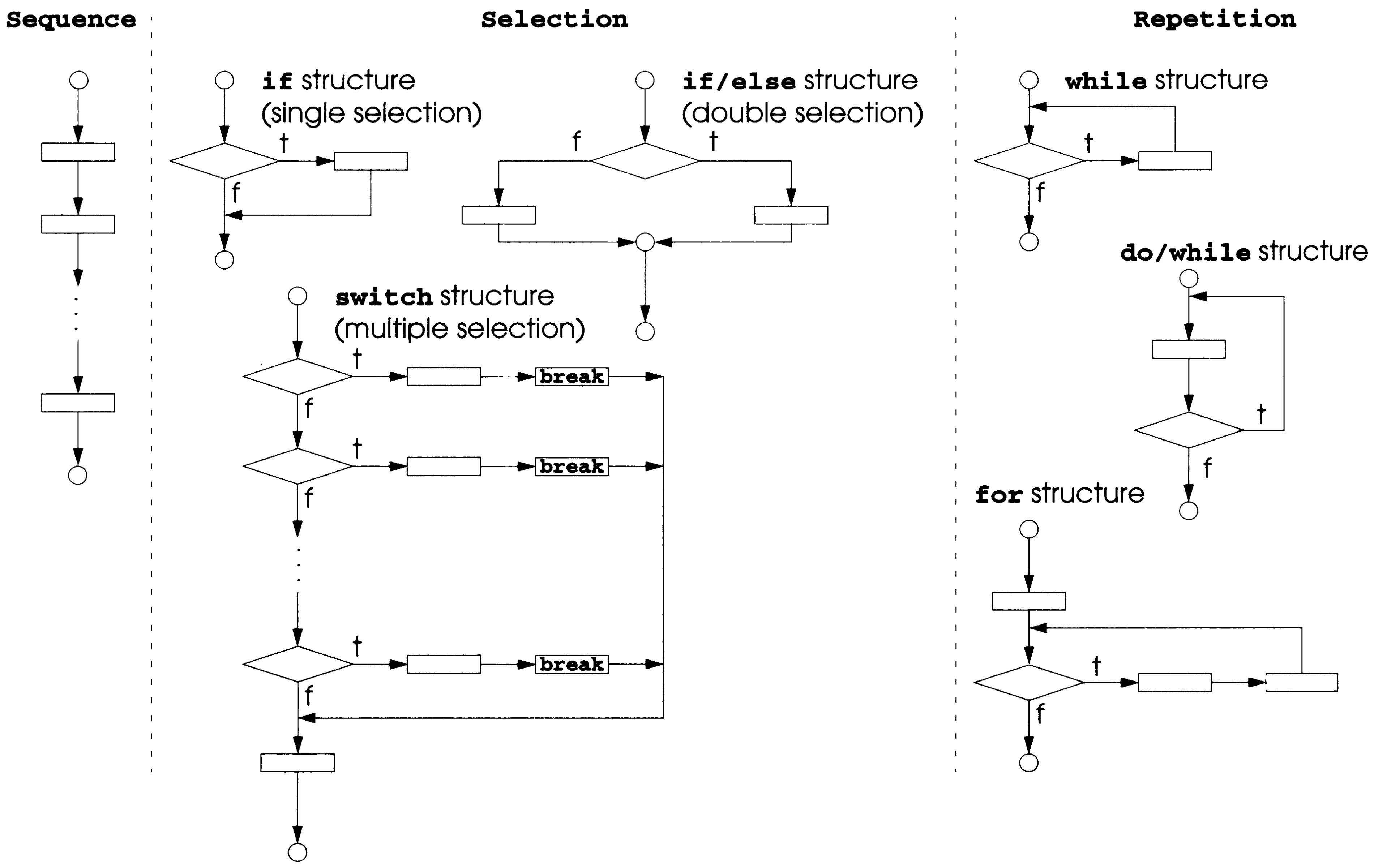
Control Structures

Control Structures

Normally, statements in a program are executed one after another in the order in which they are written. This is called sequential execution.

The transfer of control statements enable the programmer to specify that the next statement to be executed may be other than the next one in the sequence.

Sequence, Selection, and Repetition



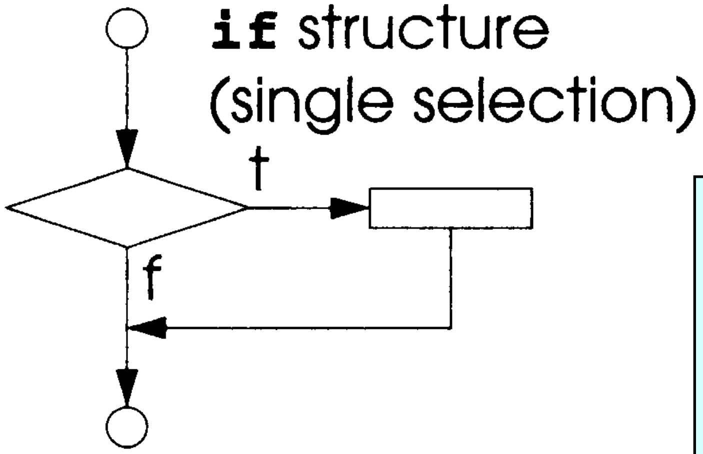
Sequence
Selection
Repetition
if structure (single selection)
if/else structure (double selection)
switch structure (multiple selection)
break
break
break
while structure
do/while structure
for structure
t
f

Three types of selection structures:

if	single-selection structure
if/else	double-selection structure
switch	multiple-selection structure

if - single-selection structure

The if selection structure performs an indicated action only when the condition is **true**; otherwise the condition is skipped

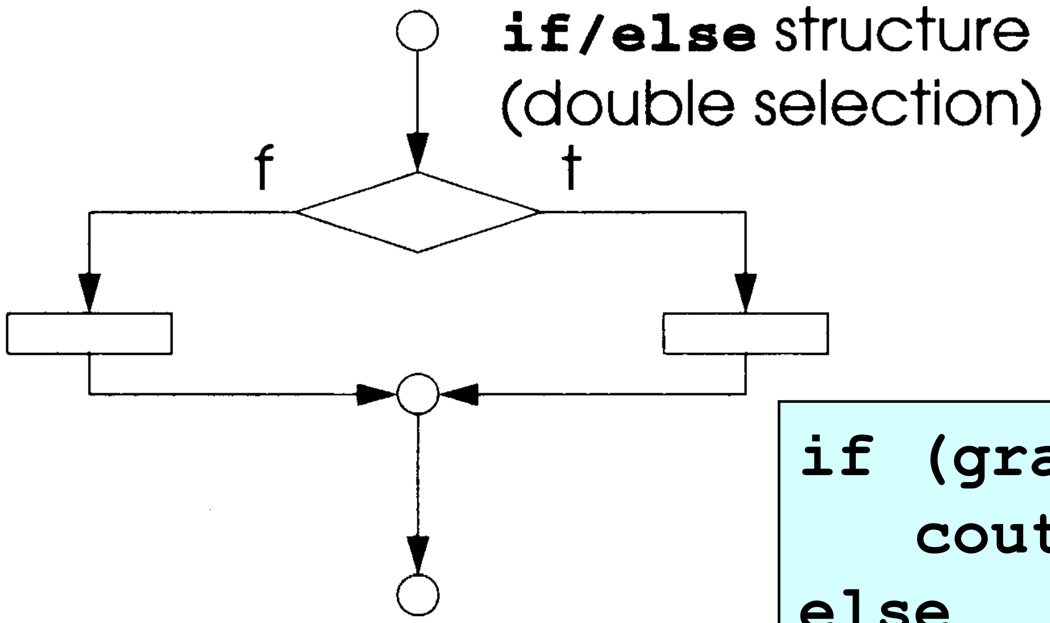


```
if (grade >=60)
   cout << "passed";

if (grade >=60) {
   n=n+1;
   cout << "passed";}
```

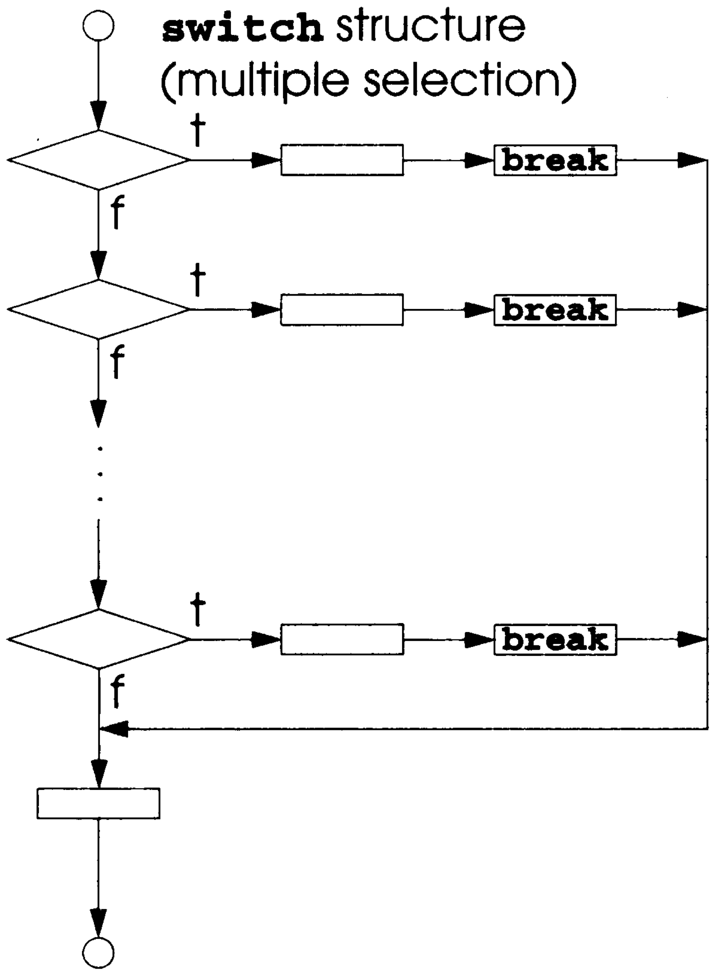
if/else - double-selection structure

The if/else selection structure allows the programmer to specify that a different action is to be performed when the condition is **true** than when the condition is **false**.



```
if (grade >=60)
   cout << "passed";
else
   cout << "failed";
```

switch - multiple-selection structure



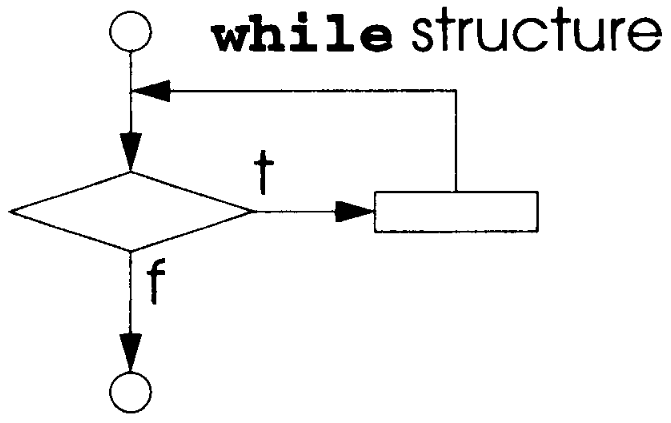
```
switch (x) {
case 1:
cout << "x is 1";
break;
case 2:
cout << "x is 2";
break;
default:
cout << "value of x
unknown";
}
```

Three types of repetition structures:

- while
- do/while
- for

The while repetition structure

A repetition structure allows the programmer to specify an action is to be repeated while some condition remains true

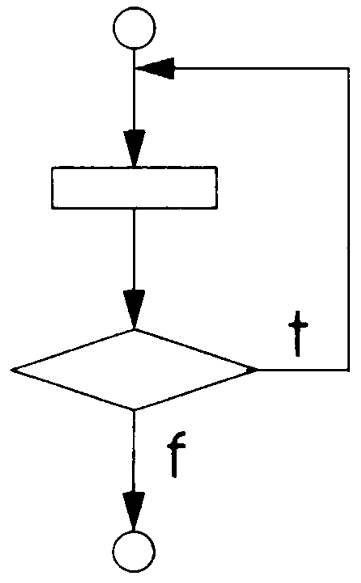


```
int n = 2;
while (n <= 100)
   {n = 2 * n;
    cout << n;}
```

The do/while repetition structure

The loop-continuation condition is not executed until after the action is performed at least once

do/while structure

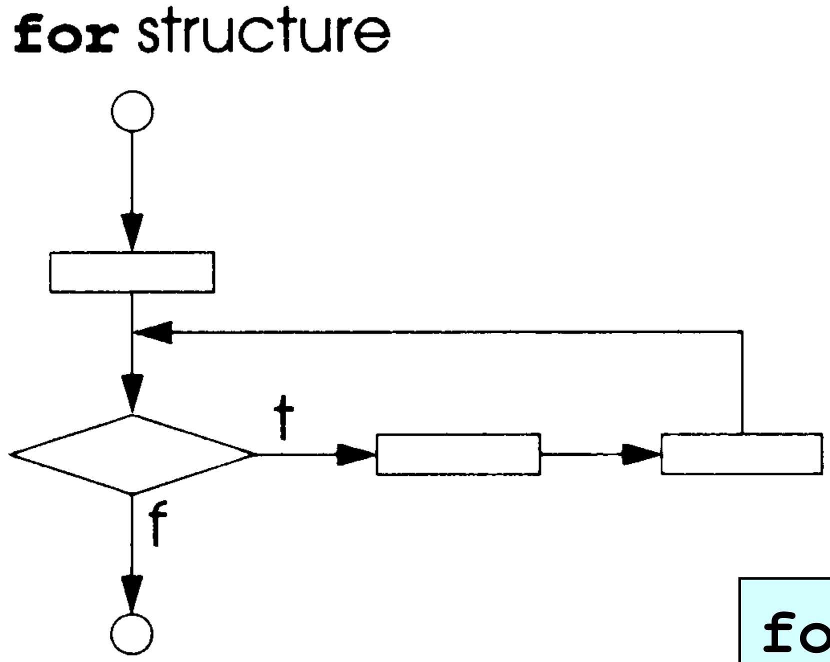


do {
   statement
} **while** (condition);

```
int i = 0;
do {
    cout << i;
    i = i + 10;
} while (i <=100);
```

The for repetition structure

The for repetition structure handles all the details of counter-controlled repetition.



```
for (i=0; i <=5; i=i+1)
{
 … actions …
}
```

The break and continue statements

The break and continue statements alter the flow of the control.

The break statement, when executed in a **while**, **for**, **do/while**, or **switch** structure, causes immediate exit from that structure

The continue statement, when executed in a **while**, **for**, or **do/while** structure, skips the remaining statements in the body of the structure, and proceeds with the next iteration.

```
// using the break statement
#include <iostream.h>
int main ()
{
int n;
for (n = 1; n <=10; n = n+1)
{
   if (n == 5)
      break;
   cout << n << "  ";
}
cout << "\nBroke out of loop at n of " << n;
return 0;
}
```

```
1  2  3  4
Broke out of loop at n of 5
```

```
// Using the continue statement
#include <iostream.h>
int main()
{
    for ( int x=1; x<=10; x++)
    {
        if (x == 5)
        {continue;}
        cout << x << " ";
    }
    cout << "\nUsed continue to skip printing
5" << endl;
    return 0;
}
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

Good practice:

The **while** structure is sufficient to provide any form of repetition.

Part 6

Functions

Functions

The best way to develop and maintain a large program is to construct it from smaller parts (modules).

Modules in C++ are called functions and classes.

C++ standard library has many useful functions.

Functions written by a programmer are *programmer-defined-functions*.

Math Library Functions

Math library functions allows to perform most common mathematical calculations

Some math library functions:

cos(x) sin(x) tan(x) sqrt (x)

exp(x) log(x) log10(x) pow(x,y)

fabs(x) floor(x) fmod(x,y) ceil(x)

Header files

Each standard library has a corresponding header file containing the function prototypes for all functions in that library and definitions of various types and constants

Examples

old styles and new styles

<math.h>	<cmath>	math library
<iostream.h>	<iostream>	input and output
<fstream.h>	<fstream>	read and write (disk)
<stdlib.h>	<cstdlib>	utility functions

… and many more

examples

old style

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <math.h>
```

new style (note – add a line)

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
using namespace std;
```

Functions prototypes

A function-prototype tells the compiler the name of the function, the type of data returned by the function, the number of parameters, the type of parameters, and the order of parameters.

Function prototype:
value-type function-name (par-type1, par-type2, …)

The compiler uses function prototypes to validate function calls.

Functions definitions

Function definition:

```
return-value-type function-name(parameter-list)
{
  declarations and statements (function body)
}
```

A type must be listed explicitly for each parameter in the *parameter-list* of a function

All variables declared in function definitions are local variables – they are known only in the function.

```
//example: a programmer-defined function
#include <iostream.h>
int square( int );   // function prototype
int main()
{
   for ( int x = 1; x <= 10; x++ )
      cout << square( x ) << "  ";
   cout << endl;
   return 0;
}
// Function definition
int square( int y )
{
   int result;
   result = y * y;
   return result;
}
```

```
1  4  9  16  25  36  49  64  81  100
```

Functions definitions

If a function does not receive any values *parameter-list* is **void** or left empty

If a function does not return any value, then return-value-type of that function is **void** both in the function prototype and function definition

```
//example: a "void" case
#include <iostream.h>
void out2(void);   // function prototype
int main()
{
   out2();
   return 0;
}
// Function definition
void out2(void)
{
   cout << "output from function out2";
   return;
}
```

```
output from function out2
```

References and Reference Parameters

There are two ways to invoke functions:

call-by-value – a copy of the argument’s value is made and passed to the called function. Changes to the copy do not affect the original variable’s value in the caller. (This this the common way)

call-by-reference – the caller gives the called function the ability to directly access the caller’s data, and to modify that data if the called function so chooses.

call-by-reference

A reference parameter is an alias for the corresponding argument.

To indicate that place & after the parameter's type in the function prototype, and the function definition.

```cpp
// call-by-reference
#include <iostream.h>
void f12(int&, int&);
int main()
{
   int a, b;
   a = 12;
   b = a;
   cout << "a = "<< a << "  b = " << b <<endl;
   f12(a, b);
   cout << "a = "<< a << "  b = " << b <<endl;
   return 0;
}
void f12(int& out1, int& out2)
{
    out1 = out1*2.0;
    out2 = out1 +3;
}
```

```
a = 12  b = 12
a = 24  b = 27
```

Default Arguments

Function calls may pass a particular value of an argument. The programmer can specify that such an argument is a *default argument* with a default value.

When a default argument is omitted in a function call, the default value is automatically inserted by the compiler and passed in the call.

Default argument must be the rightmost arguments in a function's parameter list.

Default arguments normally are specified in the prototype

```
int function2(int a=2);
```

Part 7

Arrays

Arrays

An array is a consecutive group of memory locations that all have the same name and the same type.

To refer to a particular location or element in the array, we specify the name of the array and the position number of the particular element in the array.

The first element in the every array is the 0th element.

Arrays in C/C++

Most of us were not taught by our mothers to count on our fingers starting with the thumb as zero!
Accordingly, you will probably make fewer n - 1 errors if you do not use zero subscripts when dealing with matrices.

F.S. Acton "Real Computing made real"

Declaring Arrays

Arrays occupy space in memory. The programmer specifies the type of elements and the number of elements required, so that the compiler may reserve the appropriate amount of memory.

Example: reserve 12 elements for integer array **c**

```
int c[12];
```

Example: declaration and initialization of an array **n**

```
int n[6]={2, 18, 33, 5, 21, 39};
```

```
// Initialize array a and fill with numbers
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{   int arraySize = 5;
    int i, a[ arraySize ];

    cout <<"Element"<<setw(12)<<"Value"<< endl;

    for ( i = 0; i < arraySize; i = i + 1 )
    { a[ i ] = 2 * I;
    cout <<setw(7)<<i<<setw(12)<<a[ i ]<<endl;}
return 0;
}
```

```
Element       Value
      0           0
      1           2
      2           4
      3           6
      4           8
```

Multidimensional Arrays

Example: A 2 dimensional table 3 (rows) by 5 (columns) (15 elements)

```
int toys[3][5];
```

	0	1	2	3	4
0	5	4	6	0	6
1	2	1	4	6	3
2	5	7	4	21	0

toys [2] [3] = 21;

Passing Arrays to Functions

To pass an array argument to a function, specify the name of the array without any brackets.

Example for array **time** and function **speed**.

```
float array time[24];
…
   speed( time, 24);
```

C++ passes arrays to functions using simulated *call-by-reference* – the called function **can** modify the element values in the caller’s original arrays.

```
// Passing Arrays to Functions
#include <iostream>
using namespace std;
void print_array (int [], int);
int main ()
{
     int a[] = {1, 2, 3, 4};
     int b[] = {5, 4, 3, 2, 1};
     print_array (a,4);
     print_array (b,5);
     return 0;
}
void print_array (int arg[], int length)
{
     for (int n=0; n<length; n=n+1)
     cout << arg[n] << " ";
     cout << "\n";
}
```

```
1 2 3 4
5 4 3 2 1
```

Static and Automatic Arrays

Arrays that are declared **static** are initialized when the program is loaded. If a **static** array is not explicitly initialized, that array is initialized to zero by the compiler.

In functions: static arrays contain the values stored during the previous function call. For automatic arrays it does not happen.

```
static int array_s[10];
int array_a[10];
```

```
// Static and Dynamic arrays
#include <iostream>
using namespace std;
void print_array (int [], int);
int main ()
{
     int a[5];
     static int b[5];
     print_array (a,5);
     print_array (b,5);
     return 0;
}
void print_array (int arg[], int length)
{
     for (int n=0; n<length; n=n+1)
     cout << arg[n] << " ";
     cout << "\n";
}
```

```
2147340288 4328756 1 256 1
0 0 0 0 0
```

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; }
```

```
This is the initial string content
This is a different string content
```

Getline

```
// cin with strings
#include <iostream>
 #include <string>
 using namespace std;
 int main () {
string mystr;
cout << ”Enter your first and last name \n";
getline (cin, mystr);
cout << "You entered " << "\" "<< mystr << "\" " <<".\n";
 return 0; }
```

```
Enter your first and last name
Ian Balitsky
You entered “Ian Balitsky”
```

Part 8

Input/Output with files

File processing (open and write)

To perform file processing in C++, the header files <iostream> and <fstream> must be included.

Open a file with a name “file1.txt” and write to it

```
#include <iostream>
#include <fstream>
ofstream outfile (“file1.txt”, ios::out);
…
outfile << a << endl;
```

File processing (more)

Example 2 (also works)

Open a file with a name “file2.dat” and write to it

```
#include <iostream.h>
#include <fstream.h>
ofstream outfile;

outfile.open(“file2.dat”);

outfile << a << endl;
```

File processing (open and read)

Open a file with a name “input.dat” and read from it

```
#include <iostream>
#include <fstream>
ifstream inputfile (“input.dat”, ios::in);

inputfile >> a;
```

To close a file

```
inputfile.close();
```

File open modes

Mode	Description
ios::app	Write all output to the end
ios::in	Open a file for input
ios::out	Open a file for output
ios::nocreate	If the file does not exist, the open operation fails
ios::noreplace	If the file exists, the open operation fails

Part 9

Pointers

& and * operators

When you have e.g. z = z+2 in your code, the computer

1. looks up the address that the variable z corresponds to
2. goes to that location in memory and gets the value it contains

C++ allows us to perform either one of these steps independently:

1. &z evaluates to the address of z in memory.
2. *(&z) takes the address of z and dereferences it – it retrieves the value at that location in memory.

*(&z) is the same thing as z.

Pointers

Pointers are one of the most powerful features of the C++ programming language.

Pointers are among the most difficult capabilities to master.

Pointers enable to simulate call by reference, and to create and manipulate dynamic data structures.

Declarations

Pointer variables contain memory address as their values.

Declaration:

```
int    *iPointer, i;
float  *xPointer, x;
double *zpntr;
```

To declare and initialize a pointer named ptr that points to an integer variable named x:

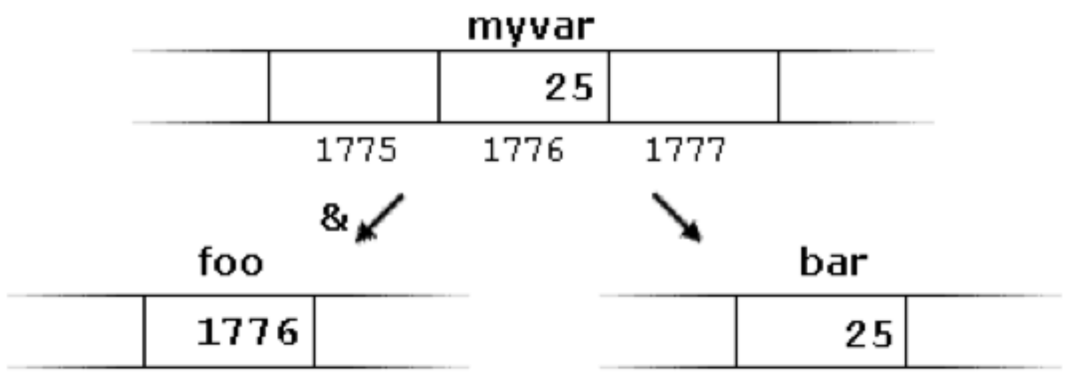
int *ptr = &x

&x gives physical address of x in the computer

Reference operator &

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

The values contained in each variable after the execution of this are shown in the following diagram:



First, we have assigned the value `25` to `myvar` (a variable whose address in memory we assumed to be `1776`).

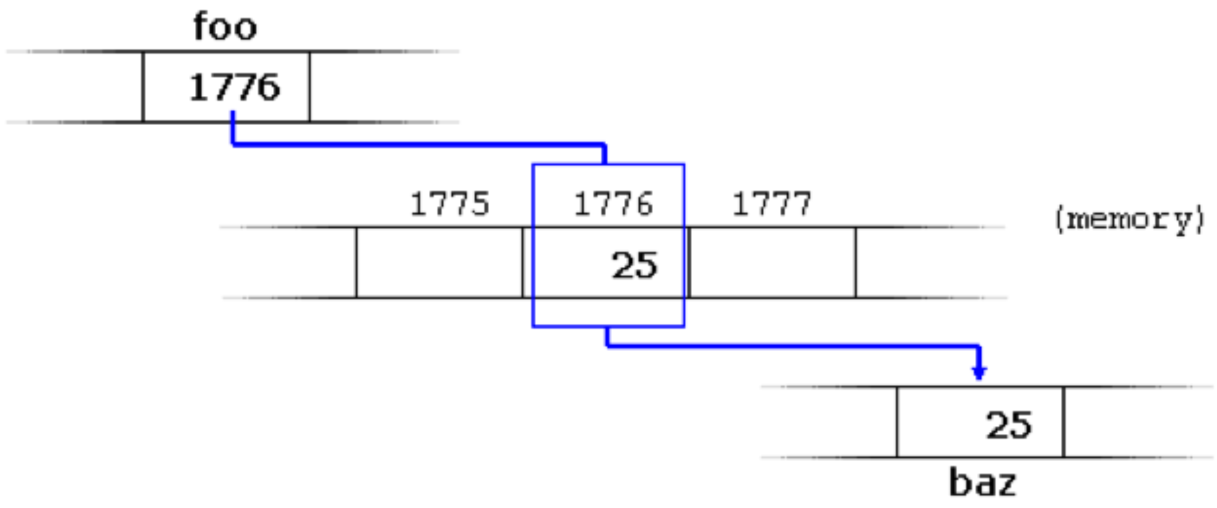
The second statement assigns `foo` the address of `myvar`, which we have assumed to be `1776`.

Finally, the third statement, assigns the value contained in `myvar` to `bar`. This is a standard assignment operation,

Dereference operator *

```
baz = *foo;
```

This could be read as: "baz equal to value pointed to by foo", and the statement would actually assign the value 25 to baz, since foo is 1776, and the value pointed to by 1776 (following the example above) would be 25.



It is important to clearly differentiate that foo refers to the value 1776, while *foo (with an asterisk * preceding the identifier) refers to the value stored at address 1776, which in this case is 25. Notice the difference of including or not including the *dereference operator* (I have added an explanatory comment of how each of these two expressions could be read):

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

Pointer operations

Important: & is *address operator* that returns the address of its operand

```
int y = 5;
int * yptr;
```

the statement

```
yptr = &y;
```

assigns the address of the variable **y** to pointer **yptr**

Now the statement

```
cout << *yptr << endl;
```

print the value of **y**, namely **5**.

And the statement

```
*yptr = 9;
```

would assign **9** to **y**.

Pointer arithmetics

Suppose now that we define three pointers in this compiler:

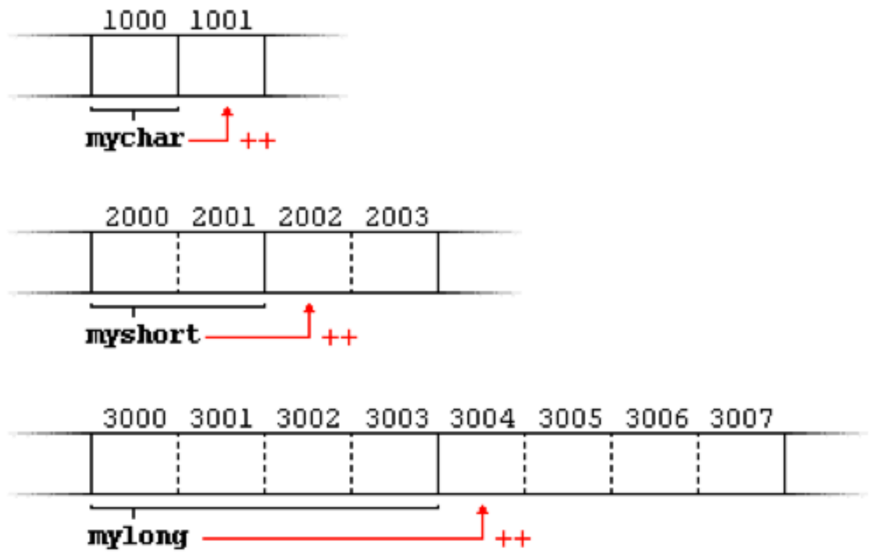
```c
char *mychar;
short *myshort;
long *mylong;
```

and that we know that they point to the memory locations `1000`, `2000`, and `3000`, respectively.

Therefore, if we write:

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

`mychar`, as one would expect, would contain the value 1001. But not so obviously, `myshort` would contain the value 2002, and `mylong` would contain 3004, even though they have each been incremented only once. The reason is that, when adding one to a pointer, the pointer is made to point to the following element of the same type, and, therefore, the size in bytes of the type it points to is added to the pointer.



Multiple uses of & and *

The * operator is used in two different ways:

1. When declaring a pointer, * is placed before the variable name to indicate that the variable is a pointer.
2. When using a pointer, * is placed to dereference it – to access or set the value it points to.

Similarly, the & operator is used

1. To indicate a reference data type (e.g. int & x)
2. To take the address of the variable (e.g. int * ptr = &x;)

```cpp
// Cube a variable using call-by-reference
// with a pointer argument
#include <iostream.h>
void cubeByReference( int * );   // prototype
int main()
{
   int number = 5;
   cout << "The side is " << number;
   cubeByReference( &number );
   cout <<"\nThe volume is "<< number << endl;
   return 0;
}
void cubeByReference( int *nPtr )
{
 *nPtr = *nPtr * *nPtr * *nPtr;//cube to main
}
```

```
The side is 5
The volume is 125
```

Function pointers

A pointer to a function contains the address of the function in memory.

A function name is the starting address in memory of the code that performs the function’s task

Pointers to functions can be processed to functions, returned to functions, stored in arrays, and assigned to other function pointers.

```
//example: using function pointers
#include <iostream.h>
float av( float, float, float(*)(float));
float x2(float);
int main()
{ float x2average, xmin, xmax;
    xmin = 2.0;
    xmax = 4.0;
    x2average = av(xmin, xmax, x2);
    cout << "average = " << x2average;
    return 0;
}

float av( float a, float b, float (*f)(float))
{ return (f(b)+f(a))/2.0;}

float x2 (float x)
{ return x*x;}
```

```
average = 10
```

Examples

```
// Example 1: calculate values of a function
//            and write to a file
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
using namespace std;

double f(double);        //function prototype

int main()
{
    const double pi=3.1415926;
    double a, b, step, x, y;
    int    i, n;
    ofstream out2disk;    //output to out2disk
```

see the next slide …

```
a = 0.0;                  //left endpoint
b = 2.0*pi;               //right endpoint
n = 12;                   //number of points

step = (b-a)/(n-1);
out2disk.open ("table01.dat");
out2disk <<"     x"<<"       f(x)"<< endl;
i=1;
while (i <= n)
{x = a + step*(i-1);
 y = f(x);
 out2disk << setw(12) << setprecision(5)
 << setiosflags(ios::fixed|ios::showpoint)
 << x << setw(12) << setprecision(5)
 << setiosflags(ios::fixed|ios::showpoint)
 << y <<endl;
```

see the next slide …

```
     i = i+1;
     }
     return 0;
}

// Function f(x)
   double f(double x)
{
     double y;
     y = sin(x);
return y;
}
```

```
     x              f(x)
     0.00000        0.00000
     0.57120        0.54064
     1.14240        0.90963
     1.71360        0.98982
     2.28479        0.75575
     2.85599        0.28173
     3.42719       -0.28173
     3.99839       -0.75575
     4.56959       -0.98982
     5.14079       -0.90963
     5.71199       -0.54064
     6.28319       -0.00000
```