C++ HELPSHEET

CHAPTER 1 – BASICS

1 STANDARD LIBRARY

About:

- Include standard C++ library

Code:

```
#include <iostream>     // Provides cout
#include <iomanip>      // Provides setw function
#include <cstdlib>      // Provides EXIT_SUCCESS
#include <cassert>      // Provides assert function
using namespace std;    // Include all standard library items
```

2 PRE-POSTCONDITION

About:

- **Precondition:** Giving condition that is required to be true when function called.
- **Postcondition:** Describes what will be true after function call is completed.

Code:

```
double celsius_to_fahrenheit(double c)
// Precondition: c is a Celsius temperature no less
// than absolute zero (-273.16).
// Postcondition: The return value is the temperature
// c converted to Fahrenheit degrees.
{
    const double MINIMUM_CELSIUS = -273.16;
    assert(c >= MINIMUM_CELSIUS);
    return (9.0 / 5.0) * c + 32;
}
```

3 ASSERT

About:

- Ensure value satisfy precondition.
- Remember to `#include <cassert>`
- To turn off assertions, `#define NDEBUG`

Code:

```
void setup_cout_fractions(int fraction_digits)
{
    assert(fraction_digits > 0);
    cout.precision(fraction_digits);
    cout.setf(ios::fixed, ios::floatfield);
    if (fraction_digits == 0)
        cout.unsetf(ios::showpoint);
    else
        cout.setf(ios::showpoint);
}
```

4 CONSTANT

About:

- Prevent value of variable being changed

Code:

```
const double TABLE_BEGIN =        -50.0;
```

5 EXIT MAIN PROGRAM

About:

- Check if main progravm exits successfully
- Defined in stdlib.h

Code:

```
return EXIT_SUCCESS;
```

6 FULL PROGRAM

Code:

```
#include <iostream>     // Provides cout
#include <iomanip>      // Provides setw function for setting output width
#include <cstdlib>      // Provides EXIT_SUCCESS
#include <cassert>      // Provides assert function
using namespace std;    // Allows all standard library items to be used

double celsius_to_fahrenheit(double c)
// Precondition: c is a Celsius temperature no less than absolute
// zero (-273.16).
// Postcondition: The return value is the temperature c converted to Fahrenheit
// degrees.
{
    const double MINIMUM_CELSIUS = -273.16; // Absolute zero in Celsius degrees
    assert(c >= MINIMUM_CELSIUS);
    return (9.0 / 5.0) * c + 32;
}

void setup_cout_fractions(int fraction_digits)
// Precondition: fraction_digits is not negative.
// Postcondition: All double or float numbers printed to cout will now be
// rounded to the specified digits on the right of the decimal.
{
    assert(fraction_digits > 0);
    cout.precision(fraction_digits);
    cout.setf(ios::fixed, ios::floatfield);
    if (fraction_digits == 0)
        cout.unsetf(ios::showpoint);
    else
        cout.setf(ios::showpoint);
}

int main( )
{
    const char   HEADING1[]  = "   Celsius"; // Heading for table's 1st column
    const char   HEADING2[]  = "Fahrenheit"; // Heading for table's 2nd column
    const char   LABEL1      =          'C'; // Label for numbers in 1st column
    const char   LABEL2      =          'F'; // Label for numbers in 2nd column
    const double TABLE_BEGIN =       -50.0; // The table's first Celsius temp.
    const double TABLE_END   =        50.0; // The table's final Celsius temp.
    const double TABLE_STEP  =        10.0; // Increment between temperatures
    const int    WIDTH       =           9; // Number chars in output numbers
    const int    DIGITS      =           1; // Number digits right of decimal pt

    double value1;  // A value from the table's first column
    double value2;  // A value from the table's second column

    // Set up the output for fractions and print the table headings.
    setup_cout_fractions(DIGITS);
    cout << "CONVERSIONS from " << TABLE_BEGIN << " to " << TABLE_END << endl;
    cout << HEADING1 << "  " << HEADING2 << endl;

    // Each iteration of the loop prints one line of the table.
    for (value1 = TABLE_BEGIN; value1 <= TABLE_END; value1 += TABLE_STEP)
    {
        value2 = celsius_to_fahrenheit(value1);
        cout << setw(WIDTH) << value1 << LABEL1 << "  ";
        cout << setw(WIDTH) << value2 << LABEL2 << endl;
    }

    return EXIT_SUCCESS;
}
```

CHAPTER 2 – C++ CLASS DESIGN

1 CONSTRUCTOR

About:

- To use constructor, declare: `point p;`
- To set values, declare: `point p(1,2);`

Code (.h):

```
class point
{
    public:
        // CONSTRUCTOR
        point();
        point(double initial_x = 0.0,
                double initial_y = 0.0);
}
```

Code (.cpp):

```
point::point()
{
    x = 0;
    y = 0;
}

point::point(double initial_x, double initial_y)
{
    x = initial_x;
    y = initial_y;
}
```

2 DESTRUCTOR (~)

About:

- Uses dynamic memory
- Returns an object's dynamic memory to the heap when object is no longer in use
- ~point();
- ** Note that usually destructor is automatically activated when object is no longer accessible

Code(.h)

```
#ifndef SRC_POINT_H_
#define SRC_POINT_H_

class point {
public:
    point();
    virtual ~point();
};

#endif /* SRC_POINT_H_ */
```

Code(.cpp):

```
#include "point.h"

point::point()
{

}

point::~point()
{

}
```

3 GETTERS AND SETTERS

About:

- **Getters:** Using constant member function, it may examine object status, but is forbidden from changing object.
- **Setters:** By setting the parameters as constant, it can prevent the parameter objects being changed (**Also known as constant reference).

Code:

```
// GETTERS
double get_x( ) const { return x; }
double get_y( ) const { return y; }
// SETTERS
void set_x(const double& x_amount);
void set_y(const double& y_amount);
```

4 INLINE MEMBER FUNCTIONS

About:

- Place function definition in class definition (header file)
- Inefficient, best is to place in implementation file

Code:

```
class point
{
    public:
        // CONSTRUCTOR
        point();
        point(double initial_x = 0.0,
                double initial_y = 0.0);
        // GETTERS
        double get_x( ) const { return x; }
        double get_y( ) const { return y; }
        // SETTERS
        void set_x(const double& x_amount);
        void set_y(const double& y_amount);
}
```

5 DEFAULT ARGUMENT

About:

- Set default value for parameters
- To override the default value, simply set value when calling the function as shown below "To use the function"

Code (Function initialization):

```
int date_check(int year, int month = 1, int date = 1);
```

Code (To use the function):

```
date_check(2016);           // month = 1, date = 1
date_check(2016, 2);        // month = 2, date = 1
date_check(2016, 2, 28);    // month = 2, date = 28
```

6 FUNCTION PARAMETERS

About:

- If the value of parameters is changed in the function, its original value:
- **Value parameters**: will not be changed after the function ends.
- **Reference parameters**: will be changed after function ends.

Code (Value):

```
void setValue_42(int i)
{
    i = 42;
}

int d = 0;
setValue_42(d);
```

**In this case, value of I after function ends still remain as default

Code (Reference):

```
void setValue_42(int& i)
{
    i = 42;
}

int d = 0;
setValue_42(d);
```

**In this case, value of I after function ends changes to 42

**Reference parameters are more preferred than value parameters as value parameters are less efficient (make extra copy to prevent value from being changed)

```
double d = 0;
setValue_42(d);
```

**If wrong argument type is set (double != int), value will still remain as default d = 0

7 BINARY COMPARISON OPERATOR OVERLOADING

About:

- **Binary function**: Involves 2 parameters
- **Types of operators**: ==, !=, <, >, <=, >=
- To compare 2 objects, we can use:

```
point p1;
point p2;

if(p1 == p2)
{
    ...
}
```

- **But if both are new classes, we cannot compare simply using == operator!

Code (.cpp):

// Overload == operator IN point class

```
bool operator == (const point& p1, const point p2)
{
    return
        ((p1.get_x() == p2.get_x())
        &&
        (p1.get_y() == p2.get_y()));
}
```

// Once operator == is defined as above, we do not need to redefine again for other operators:

```
bool operator != (const point& p1, const point p2)
{
    return !(p1 == p2);
}
```

8 BINARY ARITHMETIC OPERATOR OVERLOADING

About:

- Arithmetic operation between 2 objects

```
point operator + (const point& p1, const point p2)
{
    double x_sum, y_sum;

    x_sum = p1.get_x() + p2.get_x();
    y_sum = p1.get_y() + p2.get_y();

    point sum(x_sum, y_sum);

    // Compute sum of x and y respectively
    return sum;
}

int main()
{
    point speed1(5, 7);
    point speed2(1, 2);
    point total;

    total = speed1 + speed2;
    cout << "Sum of x: " << total.get_x();
    cout << "Sum of y: " << total.get_y();
}
```

9 I/O OPERATOR OVERLOADING

About:

- Define input and output stream of object

Code:

```
istream operator >> (istream& ins, point& p)
{
    ins >> p.x >> p.y;
    return ins;
}

ostream operator << (ostream& ins, point& p)
{
    outs << p.get_x();
    return outs;
}
```

10 FRIEND FUNCTION

About:

- Function that is not a member function, but still has access to private members of object of a class.
- ** Declared in header file

Code:

// Allow the private members to be accessed by istream to print the x & y values of point class

```
class point
{
    ...

    // FRIEND FUNCTION
    friend std::istream& operator >>
        (std::istream& ins, point& target);
}
```

CHAPTER 3 – OBJECT-ORIENTED PROGRAMMING (OOP)

1) Creating a namespace
2) Writing a header file (.h)
3) Writing the implementation file (.cpp)

1 NAMESPACE

About:

- If programmer A and B both written storage class and a program needs both storage classes
- Namespace is used to identify a portion of programmer's work
- Includes email or name of programmer
- Place in both .h and .cpp files
- ** Do not place using statement in header file!

Types:

- Make all namespace available: `using namespace programmer_name;`
- If we only need specific item from namespace: `using namespace programmer_name::storage;`
- OR just start coding without "`using namespace`": `programmer_name::storage;`

Code (in both .h and .cpp):

```
namespace programmer_name
{
    class point
    {
        ...
    }
}
```

```
namespace main_savitch_2B
{
    ...
}
```

2 HEADER FILE

About:

- Provides all information programmer needs to know for the class.
- Include header file comment on top of header file

Macro guard:

- Sometimes a class may have more than 1 header file
- Used to avoid duplicate class

Code (in .h):

```
#ifndef MAIN_SAVITCH_NEWPOINT_H
#define MAIN_SAVITCH_NEWPOINT_H
#include <iostream>

namespace programmer_name
{
    class point
    {
        ...
    };
}

#endif
```

3 IMPLEMENTATION FILE

About:

- Implement all function body from header file

Code:

// class_name::function_name

```
point::point(double initial_x, double initial_y)
{
    x = initial_x;
    y = initial_y;
}

void point::shift(double x_amount, double y_amount)
{
    x += x_amount;
    y += y_amount;
}
```

CHAPTER 4 – CONTAINER & SEQUENCE CLASSES

1 TYPEDEF & SIZE_T & CONSTANT

About:

- `typedef`: To allow for flexible value type
- `size_t`: Guarantees the values of `size_t` are sufficient to hold size of any variable declared in your machine
- `static const: static` ensures there is only one copy of this member, and `const` ensures the value cannot be changed

Code:

```
class bag
{
public:
    typedef int value_type;
    typedef std::size_t size_type;
    static const size_type CAPACITY = 30;
    ...

private:
    value_type data[CAPACITY];
    size_type used;
}
```

2 MULTISET ITERATOR

About:

- Permits programmer to easily step through all items in container, examine the items and changing them
- **begin:** First item in container
- *** operator:** Access current element
- **++ operator:** Move an iterator forward to the next item in its collection
- **end:** When iterator has reached the end, it has gone beyond the last item
- **Const iterator:** Prevent values from being changed → `multiset<int>::const_iterator it;`

Code:

```
multiset<int> ages;
multiset<int>::iterator it;
ages.insert(4);
ages.insert(10);
ages.insert(20);

for(it = ages.begin(); it != ages.end(); ++it)
{
    cout << *it << " ";
}

// Output: 4 10 20
```

CHAPTER 5 – POINTERS & DYNAMIC ARRAYS

1 POINTER

About:

- Memory of a variable

Code:

// Since address is pointing to memory of i, when value of address is changed, value of i changes

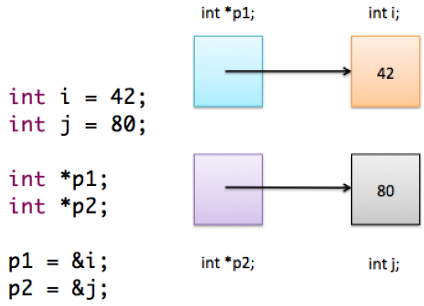
```
int *address;
int i;

i = 42;
address = &i;
*address = 0;

cout << *address << endl; // 0
cout << i << endl; // Also 0!!
```

2 POINTER ASSIGNMENT

About:

- When p2 = p1, p2 points to the same memory location as p1

Confusion about pointers:

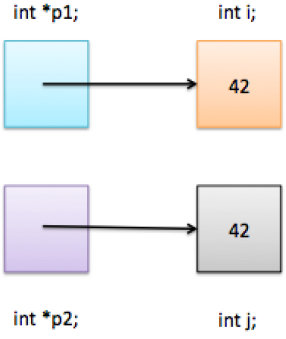


p2 = p1:

Make p2 points to the same variable p1 is already pointing to

*p2 = *p1:

Copy the value from the variable that p1 points to, to the variable p2 points to



3 DYNAMIC VARIABLES (new/delete)

About (new) operator:

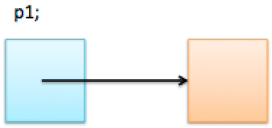
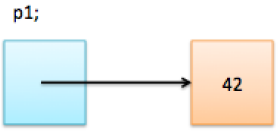
- Allocates a dynamic variable/array using the heap memory (free storage in bytes)
- Variables that are not declared and no identifier
- Created during execution of program, and only at that time the dynamic variable exists!

Code:

Data type	Object	Array

`int *ptr;`
`ptr = new int;`
	`point *ptr;`
	`ptr = new point(5)`
		`double *ptr;`
		`ptr = new double[10]`
		
		`ptr[3] = 1.23;`

E.g. Data type:

`int *p1;` *p1 is declared but is pointing to nothing*	
`p1 = new int;` *p1 is now pointing to newly allocated integer variable* *** Notice that we do not need to create an integer first before pointing p1 to an integer variable:* *// no longer needed!* *int num;* **p1 = num;*	
`*p1 = 42;` *The new integer variable now contains an integer, 42.*	

About (delete) operator:

- Release any heap memory that is no longer needed

Code:

Data type	Object	Array

`int *ptr;`
`ptr = new int;`
 	
`delete ptr;`
	`point *ptr;`
	`ptr = new point(5)`
	
	`delete ptr;`
		`int *ptr;`
		`ptr = new int[50]`
		
		`delete [] ptr;`

4 POINTER PARAMETERS

About:

- **Value parameter that is pointer:** Function may change the value the pointer points to, else cannot change!
- **Reference parameter that is pointer:** Change a pointer so that the pointer points to a new location, or changes the size of array
- **Array parameter:** Automatically treated as a pointer that points to the first element of the array

Code:

```
// Value paramter that is pointer
void function(int* i);

// Reference paramter that is pointer
void allocate_doubles(double*& p, size_t& n)
{
    cin >> n;
    p = new double[n];
}

// Array parameter
void function(int arr[]);
```

CHAPTER 6 – STL STRING CLASS

1 NULL-TERMINATED STRINGS

About:

- Special character '\0' is placed in the array immediately after the last character
- Marks the end of the string

Note:

// Since the character array contains 10 components, it can only hold 9 or fewer characters, as we need to reserve a component for '\0'

```
char s[10] = "HELLO!";
```

// In array component:

H	E	L	L	O	!	\0	?	?	?
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

** Note that if size is not given, the following allocates an array of 7 characters (including '\0')

```
char s[] = "HELLO!";
```

Code:

// Defined in cstring library

Types	Usage	Code

strcpy	Assign value to string	`char greeting[10];`
		
		`// WRONG`
		`greeting = "Hello";`
		
		`// RIGHT`
		`strcpy(greeting, "Hello");`

strcat	Concatenate string	`char[10] greeting;` `strcat(greeting, "Hello");`
NOTE: When using strcpy and strcat, if we try to copy a string with 10 characters into an array of size 5, it will write into memory locations that are not part of the array, often changing the values of declared variables!!		

strlen	Check number of characters in a string	`size_t strlen(const char s[]);`
		
		`// Example:`
		`strlen("Hello!");`

strcmp	Compares two strings	`// Returns 0 if s1 == s2`
		`// Returns -1 if s1 < s2`
		`// Returns 1 if s1 > s2`
		`int strcmp(const char s1 [],`
		`const char s2[]);`
		
		`// Exmaple:`
		`strcmp("Cool", "Lame");`

2 STRING CLASS (OBJECT)

About:

- To avoid pitfalls of null-terminated strings
- Instead of using an array of characters, a string object is used instead

Code:

Constructor	`// CONSTRUCTOR`
	`string(const char str[] = "");`
	
	`char sequence[6] = "Hello";`
	`string greeting(sequence);`
	`//OR`
	`string greeting("Hello");`

Concatenate overloading operator +=	`// OVERLOADING +=`
	`void string::operator +=(const string& addend);`
	`void string::operator +=(const char addend[]);`
	`void string::operator +=(const char addend);`
	
	`string s1;`
	`string s2("Hello");`
	
	`s1 += s2;`
	`s1 += "bye";`
	`s1 += '&';`

CHAPTER 7 – INHERITENCE

1 DERIVED CLASS

About:

- Uses inheritance
- Inherit its parent's public and protected members

Code:

// Every cukoo_clock is also an ordinary clock, all public and protected members of clock are immediately available for cukoo_clock

```
class cukoo_clock : public clock
{

}
```

2 OVERRIDE INHERITED FUNCTION

About:

- Derived class sometimes need to perform differently from the base class
- Override the base class's function

Code (.h):

```
class cukoo_clock : public clock
{
public:
    // Overriden from clock class
    int get_hour() const;
}
```

Code (.cpp):

```
// To call base class function
int clock::get hour const
{

}

// To call derived class function
int cukoo_clock::get hour const
{

}
```

REFERENCES

Michael Main & Walter Savitch (December 12, 2015). Data Structures and Other Objects Using C++ (Fourth Edition).

Google Inc. Google C++ Style Guide (January 3, 2015). Retrieved from https://google.github.io/styleguide/cppguide.html