RED HAT DEVELOPER TOOLSET

Red Hat® Developer Toolset includes the latest stable versions of the gcc compiler (C, C++) and essential complementary development tools that enhance developer productivity and improve deployment times. Available through the Red Hat Enterprise Linux® Developer Program and related Red Hat Enterprise Linux Developer Subscriptions (**redhat.com/apps/store/developers/**).

RED HAT DEVELOPER TOOLSET MAKES IT POSSIBLE TO:

- **Speed developer productivity.**
  Compile once and deploy to multiple versions of Red Hat Enterprise Linux.

- **Gain flexibility to deploy with confidence.**
  Preserve application compatibility as you deploy into production.

- **Choose the tools best suited for your project.**
  Access the annually updated toolset for the latest stable versions of essential development tools.

Learn more about the Red Hat Enterprise Linux Developer Program at **developer.redhat.com/rhel**.

Read what Red Hat and other developers are blogging about at **developerblog.redhat.com**.

Learn. Code. Share.

DZone Refcardz

Brought to you by: redhat

CONTENTS INCLUDE:

Core C++

By: Steve Oualline

C++ is a popular object-oriented programming language invented in 1979 by Bjarne Stroustrup. Originally called "C with Classes," the name was changed in 1983. C++ is used in low-level embedded systems as well as high-level, high performance services.

This Refcard is aimed at current C++ programmers, and does not fully introduce the language to beginners. The card therefore does not explain basic concepts (arrays, pointers, exceptions, etc.) in detail. For a more extensive introduction, useful for developers with any level of C++ experience, see Practical C++ Programming, by the author of this Refcard.

This Refcard follows the GCC implementation of C++.

C++ TYPES

Basic Types

Type	Description	Example
int	integer	1234, -57, 0x3E (hex), 0377 (octal)
float	floating point	0.3, 1.0e+33, -9.3
char	character	'a', 'c', '\n'
wchar_t	wide (16 bit) character (little used; UTF-8 now preferred for international characters)	L'a', L'x'
bool	true or false	true, false
void	non-existent entity (used for functions that return nothing and "typeless" pointers)	

Modifiers

Modifer	What it indicates
signed	An integer or character variable that can contain positive and negative values. (Default, rarely used)
unsigned	An integer or character variable that can contain only positive values (but twice as many as the equivalent signed number)
short	An integer variable that may contain fewer values than a normal int
long	An integer variable that may contain more values than a normal int
long long	An integer that may contain more values than a long int
double	A floating point value that may have more range and precision than a normal float
long double	A floating point variable that may have more range and precision than a double
auto	Indicate an automatically allocated stack variable. (Default. Very rarely used)
const	Indicate a variable whose value can not be changed.
register	A hint to the compiler that this variable should be placed in a register. (Rarely used and ignored by most modern compilers.)
mutable	Indicate a member variable that may be modified even in a const instance of a class

Modifer	What it indicates
volatile	Tells the compiler that this variable is an I/O port, shared memory location, or other type of memory location whose value can be changed by something other than the currently executing code
static	1. Before a normal global variable declaration: makes the scope of the variable local to the file in which it's declared. 2. Modifying a function: makes the scope of the function locate to the file in which it's declared. 3. Inside a function: indicates a variable that is initialized once and whose value does not change if the function returns and is called again. 4. Member variable declaration: indicates a variable that belongs to the class itself instead of an instance of the class. In other words, one variable will be used for all classes. 5. Member function declaration: indicates a function that belongs to the class itself and not an instance of the class. This type of function can only access static member variables of the class. 6. Constant member variable: indicates that a constant belongs to the class itself and not an instance of the class.
extern	Indicates that a function or variable is potentially declared in another module

To change an expression from one type to another, see 'Casts' below.

NAMESPACES

Declaration	What it does
namespace name {...}	Define namespace for the enclosed code.
using name;	Import function and variable definition from the given namespace into the current namespace. (Rarely used -- many prefer to use the fully scoped names. e.g. std::cout instead of using std; cout.)

CONST POINTER DECLARATION

Pointers, like variables, can be declared constant. The value can also be declared constant. Both can be declared constant at the same time, like this:

char* foo;	Pointer can be modified. Data can be modified.
const char* foo;	Pointer can be modified. Data can be **not** modified.
char* const foo;	Pointer can be **not** modified. Data can be modified.
const char* const foo;	Pointer can be **not** modified. Data can **not** be modified.

CASTS

C++ is strongly typed. This means that most type conversions must be done explicitly. This conversion process is called casting. To cast:

const_cast	Redefine the "const" value of a variable.
static_cast	Change the type of a variable. This is a compile time operation and an incorrect cast will not result in a runtime error.
reinterpret_cast	Used for non-portable casting of one bit pattern to another.
dynamic_cast	Safely cast variable from one type to another. If the cast can not be made, throw a std::bad_cast exception.

PRE-PROCESSOR DIRECTIVES

Each pre-processor directive begins with **#** and ends with a newline. If you want to split a directive onto two lines, precede the newline with **** .

Directive	What it does
#include <file>	Include a system header file in the program
#include "file"	Include a local header file in the program
#define NAME value	Define a macro. (In most cases a const declaration is better than a #define.)
Note: When the value is an expression, it is a good idea to put parentheses around the entire value, e.g.: #define AREA (width * height)	
#define MACRO(p1, p2) \ value	Define a parameterized macro. (In almost all cases creating an inline function is preferable to a parameterized macro.)
Note: It is a good idea to put () around the parameters when defining a macro, e.g.: #define SQUARE(x) ((x) * (x)) size = square(3+5)	
#undef NAME	Undefine a previously defined macro
#ifdef NAME	Compile the following code if NAME is defined
#ifndef NAME	Compile the following code if NAME is not defined
#else	Reverse the previous #ifdef, #ifndef, or #if
#endif	End of a #ifdef, #ifndef, or #if block
#if <expression>	Compile the following code if the expression is true
#error Message	Raise a compile time error
#warning Message	Raise a compile time warning
#line <line-info>	Synchronize line counters. Used by programs that produce C++ code so that the compiler can issue error message using the original file's line number.
#pragma <system-dependent>	Used for compiler dependent settings and operations

OPERATOR PRECEDENCE

Precedence	Operator	Description
1	: :	scope resolution
2	++ --	suffix increment/decrement
	()	function call
	[]	array subscripting
	.	element selection (reference)
	->	element selection (pointer)
3	Typeid const_cast dymanic_cast reinterpret_cast	
4	static_cast	
5	++ --	prefix increment/decrement
	+ -	unary plus/minus
	! ~	logical / bitwise NOT
	(type)	C-style cast
	*	pointer dereference
	&	address of
	sizeof new delete	
6	.* ->*	pointer to member
7	* / %	multiply/divide/remainder
8	+ -	add/subtract
9	<< >>	bitwise shift left/right
10	< <= > >=	(relational) < / ≤ / > / ≥
11	== !=	(relational) = / ≠
12	& ^	bitwise AND/XOR
13	|	bitwise OR
14	&&	logical AND
15	||	logical OR
16	?:	ternary conditional
17	=	direct assignment
	+= -+	assignment by sum/difference
	*= /= %=	assignment by product/quotient/remainder
	<<= >>=	assignment by bitwise shift left/right
	&= ^= |=	assignment by bitwise AND/OR/XOR
18	throw	throw (exception)
19	,	comma

FUNCTION DECLARATION

The basic format of a function is:

```
type name ( parameter1, parameter2, ...) { statements }
```

Type	Indicates
void	Indicates a function that does not return a value. void resetAll() {...}
static	Scope declaration (see variable modifiers above) static int getDate(void) {...}
inline	A hint to the compiler, telling it that this function is small enough that body of the code for this function can be inserted inline rather than using the normal call / return instruction sequence. static inline square(const int x) { return (x*x); }
extern	Function that may be defined in another compilation unit. extern int getTime(void);

Type	Indicates
extern "C"	Indicates an external function that uses the C language calling conventions extern "C" time_t time(time_t* ptr)

Parameter Passing

Pass by value: the default way of passing. Changes to parameter are not passed back to the calling code.

```
void funct(int parameter)
```

Pass by reference: a reference is made to the original value. Changes to the parameter are passed back to the original code.

```
void funct(int& parameter)
```

Pass by constant reference: same as passing by reference only changes are not allowed. This is an efficient way of passing in large structures and classes.

```
void funct(const int& parameter)
```

Pass by pointer: a pointer parameter is passed by value. Although the pointer may not be changed in the original code the data pointed to can.

Note: The above function does not modify the value of parameter. The function is better written as:

```
void funct(int* const parameter) {
    *parameter = 5;
}
```

Note: Pass by reference is preferred where a single value is being passed. Legacy C code frequently uses this form of parameter passing.

Default Arguments

Arguments at the end of the arguments list can have default values, assigned like this:

```
void func(int i, int j=2, float k=3.0);
```

Variable Argument List

Functions can have variable numbers of arguments. For example, you might want to write a function to calculate the average temperature of an unknown number of fluid-samples.

To do this, use the **va_list** macro. Here's how va_list breaks down:

Element	What it does
va_list variable	Type definition for a variable list
va_start(arg_list, parameter)	Identify the start of a variable argument list. The parameter is the last variable in the fixed argument list.
va_arg(n, type)	Get an argument from the list
va_end	Indicate that argument processing is complete

Definitions for variable argument lists are in the header file **cstdarg**. Here's how that works out in code:

```
#include <cstdarg>

void funct(int num, ... )
{
    va_list arg_list;
    va_start(arg_list, num);
    int i = va_arg (arg_list, int);
    int j = va_arg(arg_list, double);
      va_end(arg_list);
}
```

Automatic Conversion

Array parameters are automatically converted to pass by pointer.

```
void funct(int* parameter) //equivalent to
void funct(int parameter[])
int array[5]

funct(array) //equivalent to
funct(&array[0])
```

Warning: Dangling References

Dangling references occur when a reference to a local variable or parameter is returned. In this example, a reference to **result** is returned. However, the variable **result** goes out of scope when the function is returned and destroyed. Any code which uses this function will get a reference to a destroyed variable.

```
int& bad_function(void) {
          int result = 5;

          return (result); //error!
}
```

CLASS DEFINITION

The generic form of a class definition:

```
class name: base class specification {
    public:
          <public members>;
    private:
          <private members>;
    protected:
          <protected members>;
};
```

The public, private, and protected members sections may be repeated as many times as needed.

Member Protections

public	Anyone outside the class may access these member functions and variables.
private	Only the class's member functions and friends may access the data.
protect	Only the class's member functions, friends, and derived classes may access.

The default protection is **private**, although good style practice dictates that you always specify the protection explicitly.

Member Modifiers

const	Cannot be changed
static	Belongs to the class, not an instance of the class
mutable	A member that may be changed even if the class cannot
const	(After function definition) -- A member function that can be called in a constant instance of the class
explicit	A constructor that must be explicitly specified. Prevents automatic type conversions during initialization.
virtual	Indicate a member function that can be overridden by a derived class that uses this one as a base.
virtual = 0	Pure virtual function. A member function that must be overridden by a derived class.

The Big 4 Member Functions

```
class foo {
    public:

          //default constructor
          void foo(void);

          //copy constructor
          void foo(const foo& other);

          //assignment operator
          foo& operator=(const foo& other);

          //destructor
          ~foo(void);
};
```

If these are not defined in your code, C++ will supply defaults.

Note: if this class is to be used as a base for other classes, the destructor should be virtual.

Base Class Specification

```
class derived: public base
```

(In this case, **derived** is the derived class and **base** is the base class.)

The access protection **public** indicates that all public members of the base class are available to users of the derived class.

```
class derived: protected base
```

Only classes derived from **derived** can access **base**.

```
class derived: private base
```

Members of **base** cannot be accessed.

```
class derived: virtual public base
```

Virtual base classes allow for consolidation of common base classes when doing multiple derivations.

The rest of this card collects information on some of the most common methods for output stream manipulation, with special focus on strings. (For a similar treatment of Java, see the Core Java Refcard: http://refcardz.dzone.com/refcardz/core-java). Some of the objects mentioned (e.g. iterators) can also be used with other data types.

OUTPUT STREAM MANIPULATION

std::endl	Writes a newline and flushes output
std::ends	Writes a null character ('\0') and flushes output
std::flush	Flushes output
std::resetiosflags(ios_base::fmtflags mask)	Reset the given ioflags (see below)
std::setiosflags (ios_base::fmtflags mask)	Set the given ioflags (see below)
std::setbase(int base)	Set the base for output conversion
std::setfill(char c)	Set the fill character
std::setprecision(int n)	Set the precision for floating point output
std::setw(int n)	Set the width for numeric output
std::boolalpha	Print true and false as characters
std::noboolalpha	Print true and false as 1 and 0
std::showbase	Print a prefix (0x or 0) for hexadecimal or octal numbers
std::noshowbase	Do not output a base prefix
std::showpoint	Always show a decimal point for floating point output
std::noshowpoint	Do not show a decimal point if a floating point number has no factional values
std::showpos	Put a "+" front of positive numbers
std::noshowpos	Put nothing in front of positive numbers
std::skipws	When reading, skip whitespace until a number or other item is found
std::noskipws	Do not skip whitespace when reading items
std::uppercase	When writing items (other than character-based items) that require letters, use upper-case letters.
std::nouppercase	When writing items (other than character-based items) that require letters, use lower-case letters.
std::unitbuf	Flush output after each "unit" (line) is written
std::nounitbuf	Flush output when the buffer fills up
std::internal	Pad output by adding a fill character internally
std::left	Left justify output by padding the right side
std::right	Right justify output by padding the left side
std::dec	Output numbers in decimal format
std::hex	Output numbers in hexadecimal format
std::oct	Output numbers in octal format
std::fixed	Output floating point numbers in fixed point format
std::scientific	Output floating point numbers in scientific notation

I/O Flags

std::ios::boolalpha	If set, write boolean values as characters
std::ios::showbase	Write a prefix to show the base (hex, octal, decimal) being output
std::ios::showpoint	Output floating point numbers with a decimal point
std::ios::showpos	Put a "+" in front of positive values
std::ios::skipws	Skip whitespace on input
std::ios::unitbuf	Flush output after each unit is written
std::ios::uppercase	Use upper case letters for numerical output (1E33, 0xABC)
std::ios::dec	Output numbers in decimal format
std::ios::hex	Output numbers in hexadecimal format
std::ios::oct	Output numbers in octal format
std::ios::fixed	Output floating point numbers in fixed format
std::ios::scientific	Output floating point numbers in floating point format
std::ios::internal	Pad by adding the fill character between the sign and the number
std::ios::left	Left justify the output
std::ios::right	Right justify the output

I/O Open Flags

std::ios::app	Append data to the file
std::ios::ate	Seek to the end at opening
std::ios::binary	Binary output
std::ios::in	Open the file for input
std::ios::out	Open the file for output
std::ios::trunc	Truncate the file upon opening

STD::STRING

Type Definitions

Type	Description
value_type	Type of values used by the string. In this case char.
size_type	Type used to hold size of the string
difference_type	Type which can hold the difference between two size_type variables
reference	A reference to the data in the string (char)
const_reference	Constant reference to the data in the string.
pointer	Pointer to a character
const_pointer	Pointer to a constant character
iterator	Random access iterator for moving through the string
const_iterator	Iterator whose referenced value can not be changed
reverse_iterator	Iterator designed to move backward through the string
const_reverse_iterator	Reverse iterator with constant data

Constants

npos	"Position" returned by functions to indicate that no position was located

Constructors and Assignment

string()	Construct an empty string
string(std::string& str)	Copy constructor. Copies initial value from an existing string.
string(std::string& str, size_type pos)	Initialize with substring of str starting at pos
string(std::string& str, size_type pos, size_type length)	Initialize with an initial substring of str starting at pos and going for length characters
string(const char* const c_str)	Initialize with a C style string
string(const char* const c_str, size_type len)	Initialize with the first len characters from a C style string
string(size_type count, char ch)	Create a string by repeating ch count times
string(iterator begin, iterator end)	Create a string by copying the string between the two iterators starting at begin and ending just before end
operator = (const std::string str)	Assignment operator
operator = (const char* const c_str)	C style string, assignment operator
operator = (char ch)	Single character assignment

Other Member Functions

iterator begin()	Return an iterator pointing to the first character of the string
const_iterator begin() const	Return a const_iterator pointing to the first character of the string
iterator end()	Return an iterator that points one past the end of the string
const_iterator end() const	Return a const_iterator that points one past the end of the string
reverse_iterator rbegin()	Return a reverse_iterator pointing to the last character of the string
const_reverse_iterator rbegin() const	Return a const_reverse_iterator pointing to the last character of the string
reverse_iterator rend()	Return a reverse_iterator pointing one before the first character of the string
const_reverse_iterator rend() const	Return a const_reverse_iterator pointing one before the first character of the string
size_type size()	Return the number of characters in the string (does not include null termination)
size_type length()	Same as size()
size_type capacity()	Return the number of characters that can fit in the string before additional memory must be allocated
size_type max_size()	Return the number of bytes allocated for storage of the string.
resize(size_type size)	Resize the string. If extra characters are needed, a null character (\0) is used.
resize(size_type size, char ch)	Resize the string. If extra characters are needed, the given character will be used.
reserve(size_type size = 0)	Attempts to reserve memory so the string can hold the specified number of characters
clear()	Remove all data from the string
bool empty()	Return true if the string is empty
char operator[size_type pos]	Return the character at the given position. **Dangerous.** If the character does not exist, the behavior is undefined.
char at(size_type pos)	Return the character at the given position. If there is no such character, throw an std::out_of_range error.
operator += (std::string& str)	Append a string on to this one
operator += (const char* c_str)	Append a C style string to this string
operator += (char ch)	Append a character to this string
append(std::string str)	Append a string to this one
append(std::string str, size_type pos, size_type n)	Append a substring of str to this one
append(const char* const c_str)	Append a C style string to this one
append(const char* const c_str, size_type len)	Append a C style string to this one with limited length
append(size_type count, char ch)	Append a string consisting of repeating ch count times
append(iterator first, iterator last)	Append a string specified by iterators to this one
push_back(char ch)	Append a single character to the string
assign(std::string str)	Replace the string with a new one
assign(std::string size_type pos, size_type length)	Assign this string the value of the given substring
assign(const char* const c_str)	Replace the string with the given C style string
assign(const char* const c_str, size_type length)	Replace the string with the given C style string with limited length

assign(size_type count, char ch)	Replace the string with one containing ch repeated count times
assign(iterator begin, iterator end)	Assign the string the value of the string delimited by the iterators
insert(iterator where, iterator begin, iterator end)	Insert string specified by begin, end just before where
insert(size_type where, const std::string& str)	Insert the given string just before where
insert(size_type where, const std::string& str, size_type start, size_type length)	Insert the given string starting at start and going for length characters just before where
insert(size_type where, const char* const c_str)	Insert the C style string just before where
insert(size_type where, const char* const c_str, size_type length)	Insert the C style string for length characters just before the position where
insert(size_type where, size_type count, char ch)	Insert a string made of ch characters repeated count times just before where
insert(iterator where, char ch)	Insert a single character just before where. After all of the above, it hardly seems worth it.
string = erase()	Erase the entire string. Returns a reference to the string.
string = erase(size_type pos)	Erase the string from pos to the end. Returns a reference to the string.
string = erase(size_type pos, size_type length)	Erase the string from pos for length characters. Returns a reference to the string.

iterator erase(iterator where)	Erase one character. Returns the iterator where which points to the character after the erasure.
iterator = erase(iterator start, iterator last)	Erase from first to just before last. Returns the location of the character just after the last removal.
string = replace(size_type where, size_type length, const std::string& str)	Replace characters starting at where and continuing for length with the given replacement string. Returns a reference to the new string.
string = replace(size_type where, size_type length, const std::string& c_str, size_type replace_where, size_type replace_length)	Replace characters starting at where and continuing for length with the given replacement string starting at replace_where for replace_length. Returns a reference to the new string.

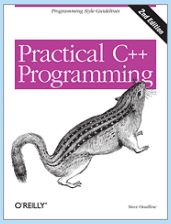
ADDITIONAL RESOURCES

- **Standard Template Library**: http://www.cplusplus.com/reference/stl/
- **GNU Compiler Collection**: http://gcc.gnu.org/
- **C++ spec (working draft)**: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf
- **Standard C++ Foundation** (director is Bjarne Stroustrup): http://isocpp.org/
- **General C++ forum** (very active): http://www.cplusplus.com/forum/general/

ABOUT THE AUTHOR

Steve Oualline has been programming for more than 45 years. He is the author of numerous books, including "Practical C++ Programming", "How Not to Program in C++", and "Wicked Cool Perl". Currently he works as a programmer for Linear Corp. and spends his spare time playing with the trains at the Orange Empire Railroad Museum.

RECOMMENDED BOOK



C++ is extremely powerful, and experienced C++ programmers can develop complex applications very efficiently. But mapping generic object-oriented programming ability, on the one hand, or facility with C, on the other, to the complexities of C++, can be difficult. This book covers all aspects of C++ programming, from software engineering to debugging, to common mistakes and how to fix them.

BUY NOW

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. "DZone is a developer's dream," says PC Magazine.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-80-6
ISBN-10: 1-936502-80-1
50795
9 781936 502806
$7.95

Version 1.0