

C++ Reference Card

© 2002 Greg Book

Key

switch – keyword, reserved
"Hello!" – string
// comment – commented code
close() – library function
main – variable, identifier
variable – placeholder in syntax
if (expression) – syntax
 statement;

Identifiers

These are ANSI C++ reserved words and cannot be used as variable names.

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t

Data Types

Variable Declaration

special class size sign type name;
special: **volatile**
class: **register, static, extern, auto**
size: **long, short, double**
sign: **signed, unsigned**
type: **int, float, char (required)**
name: the variable name (required)
// example of variable declaration
extern short unsigned char AFlag;

TYPE	SIZE	RANGE
char	1	signed -128 to 127
		unsigned 0 to 255
short	2	signed -32,768 to 32,767
		unsigned 0 to 65,535
long	4	signed -2,147,483,648 to 2,147,483,647
		unsigned 0 - 4,294,967,295
int	varies depending on system	
float	4	3.4E +/- 38 (7 digits)
double	8	1.7E +/- 308 (15 digits)
long double	10	1.2E +/- 4,932 (19 digits)
bool	1	true or false
wchar_t	2	wide characters

Pointers

type *variable; // pointer to variable
type *func(); // function returns pointer
void * // generic pointer type
NULL; // null pointer
*ptr; // object pointed to by pointer
&obj // address of object

Arrays

int array[n]; // array of size n
int array2[n][m]; // 2d n x m array
int array3d[i][j][k]; // 3d i x j x k array

Structures

```
struct name {  
    type1 element1;  
    type2 element2;  
    ...  
} object_name; // instance of name  
name variable; // variable of type name  
variable.element1; // ref. of element  
variable->element1; // reference of  
pointed to structure
```

Initialization of Variables

```
type id; // declaration  
type id, id, id; // multiple declaration  
type *id; // pointer declaration  
type id = value; // declare with assign  
type *id = value; // pointer with assign  
id = value; // assignment
```

Examples

```
// single character in single quotes  
char c='A';  
// string in double quotes, ptr to string  
char *str = "Hello";  
int i = 1022;  
float f = 4.0E10; // 4^10  
int arry[2] = {1,2}; // array of ints  
const int a = 45; // constant declaration  
struct products { // declaration  
    char name [30];  
    float price;  
};
```

```
products apple; // create instance  
apple.name = "Macintosh"; // assignment  
apple.price = 0.45;  
products *pApple; // pointer to struct  
pApple->name = "Granny Smith";  
pApple->price = 0.35; // assignment
```

Exceptions

```
try  
// code to be tried... if statements  
statements;  
// fail, exception is set  
throw exception;  
}  
catch (type exception) {  
// code in case of exception  
statements;  
}
```

C++ Program Structure

```
// my first program in C++  
#include <iostream.h>  
int main ()  
{  
    cout << "Hello World!";  
    return 0;  
}  
  
// single line comment  
/* multi-line  
comment */
```

Operators

priority/operator/desc/ASSOCIATIVITY

1	::	scope LEFT
2	()	parenthesis LEFT
	[]	brackets LEFT
	->	pointer reference LEFT
	.	structure member access LEFT
	sizeof	returns memory size LEFT
3	++	increment RIGHT
	--	decrement RIGHT
	~	complement to one (bitwise) RIGHT
	!	unary NOT RIGHT
	&	reference (pointers) RIGHT
	*	dereference RIGHT
	(type)	type casting RIGHT
	+	unary less sign RIGHT
4	*	multiply LEFT
	/	divide LEFT
	%	modulus LEFT
5	+	addition LEFT
	-	subtraction LEFT
6	<<	bitwise shift left LEFT
	>>	bitwise shift right LEFT
7	<	less than LEFT
	<=	less than or equal LEFT
	>	greater than LEFT
	>=	greater than or equal LEFT
8	=	equal LEFT
	!=	not equal LEFT
9	&	bitwise AND LEFT
	^	bitwise NOT LEFT
		bitwise OR LEFT
10	&&	logical AND LEFT
		logical OR LEFT
11	?	conditional RIGHT
12	=	assignment
	+=	add/assign
	-=	subtract/assign
	*=	multiply/assign
	/=	divide/assign
	%=	modulus/assign
	>>=	bitwise shift right/assign
	<<=	bitwise shift left/assign
	&=	bitwise AND/assign
	^=	bitwise NOT/assign
	=	bitwise OR/assign
13	,	comma

User Defined DataTypes

```
#typedef existingtype newtypename;  
#typedef unsigned int WORD;  
enum name{val1, val2, ...} obj_name;  
enum days_t {MON,WED,FRI} days;  
union model_name {  
    type1 element1;  
    type2 element2; ...  
} object_name;  
union mytypes_t {  
    char c;  
    int i;  
} mytypes;  
struct packed { // bit fields  
    unsigned int flagA:1; // flagA is 1 bit  
    unsigned int flagB:3; // flagB is 3 bit  
}
```

Preprocessor Directives

```
#define ID value // replaces ID with  
//value for each occurrence in the code  
#undef ID // reverse of #define  
#ifdef ID //executes code if ID defined  
#ifndef ID // opposite of #ifdef  
#if expr // executes if expr is true  
#else // else  
#elif // else if  
#endif // ends if block  
#line number "filename"  
// #line controls what line number and  
// filename appear when a compiler error  
// occurs  
#error msg //reports msg on cmpl. error  
#include "file" // inserts file into code  
// during compilation  
#pragma //passes parameters to compiler
```

Control Structures

Decision (if-else)

```
if (condition) {  
    statements;  
}  
else if (condition) {  
    statements;  
}  
else {  
    statements;  
}  
if (x == 3) // curly braces not needed  
flag = 1; // when if statement is  
else // followed by only one  
flag = 0; // statement  
Repetition (while)  
while (expression) { // loop until  
    statements; // expression is false  
}  
Repetition (do-while)  
do {  
    statements; // as long as condition  
} while (condition); // is true  
Repetition (for)  
init - initial value for loop control variable  
condition - stay in the loop as long as condition  
is true  
increment - change the loop control variable  
for (init; condition; increment) {  
    statements;  
}
```

```
Bifurcation (break, continue, goto, exit)  
break; // ends a loop  
continue; // stops executing statements  
// in current iteration of loop cont-  
// inues executing on next iteration  
label:  
goto label; // execution continues at  
// label  
exit(retcode); // exits program  
Selection (switch)  
switch (variable) {  
    case constant1: // chars, ints  
        statements;  
        break; // needed to end flow  
    case constant2:  
        statements;  
        break;  
    default:  
        statements; // default statements  
}
```

Console Input/Output

[See File I/O on reverse for more about streams]

C Style Console I/O

```
stdin - standard input stream  
stdout - standard output stream  
stderr - standard error stream  
// print to screen with formatting  
printf("format", arg1,arg2,...);  
printf("nums: %d, %f, %c", 1,5.6,'C');  
// print to string s  
sprintf(s, "format", arg1, arg2,...);  
sprintf(s, "This is string # %i",2);  
// read data from keyboard into  
// name1,name2,...  
scanf("format",&name1,&name2, ...);  
scanf("%d,%f",var1,var2); // read nums  
// read from string s  
sscanf("format",&name1,&name2, ...);  
sscanf(s,"%i,%c",var1,var2);
```

C Style I/O Formatting

```
%d, %i integer  
%c single character  
%f double (float)  
%o octal  
%p pointer  
%u unsigned  
%s char string  
%e, %E exponential  
%x, %X hexadecimal  
%n number of chars written  
%g, %G same as f for e,E
```

C++ console I/O

```
cout<< console out, printing to screen  
cin>> console in, reading from keyboard  
cerr<< console error  
clog<< console log  
cout<<"Please enter an integer: ";  
cin>>i;  
cout<<"num1: "<<i<<"\n"<<endl;
```

Control Characters

```
\b backspace \f form feed \r return  
\ ' apostrophe \n newline \t tab  
\nnn character #nnn (octal) \" quote  
\NN character #NN (hexadecimal)
```

Character Strings

The string "Hello" is actually composed of 6 characters and is stored in memory as follows:
Char H e l l o \0
Index 0 1 2 3 4 5
\0 (backslash zero) is the null terminator character and determines the end of the string. A string is an array of characters. Arrays in C and C++ start at zero.
str = "Hello";
str[2] = 'e'; // string is now 'Heelo'
common <string.h> functions:
strcat(s1,s2) strcat(s1,c) strcpy(s1,s2)
strcpy(s2,s1) strlen(s1) strncpy(s2,s1,n)
strstr(s1,s2)

Functions

In C, functions must be prototyped before the main function, and defined after the main function. In C++, functions may, but do not need to be, prototyped. C++ functions must be defined before the location where they are called from.

```
// function declaration  
type name(arg1, arg2, ...) {  
    statement1;  
    statement2;  
    ...  
}  
type - return type of the function  
name - name by which the function is called  
arg1, arg2 - parameters to the function  
statement - statements inside the function  
// example function declaration  
// return type int  
int add(int a, int b) { // parms  
    int r; // declaration  
    r = a + b; // add nums  
    return r; // return value  
}
```

function call

```
num = add(1,2);
```

Passing Parameters

Pass by Value

```
function(int var); // passed by value  
Variable is passed into the function and can be changed, but changes are not passed back.
```

Pass by Constant Value

```
function(const int var);  
Variable is passed into the function but cannot be changed.
```

Pass by Reference

```
function(int &var); // pass by reference  
Variable is passed into the function and can be changed, changes are passed back.
```

Pass by Constant Reference

```
function(const int &var);  
Variable cannot be changed in the function.
```

Passing an Array by Reference

```
It's a waste of memory to pass arrays and structures by value, instead pass by reference.  
int array[1]; // array declaration  
ret = aryFunc(&array); // function call  
int aryFunc(int *array[1]) {  
    array[0] = 2; // function  
    return 2; // declaration  
}
```

Default Parameter Values

```
int add(int a, int b=2) {  
    int r;  
    r=a+b; // b is always 2  
    return (r);  
}
```

Overloading Functions

Functions can have the same name, and the same number of parameters as long as the parameters are of different types
// takes and returns integers
int divide (int a, int b)
{ return (a/b); }
// takes and returns floats
float divide (float a, float b)
{ return (a/b); }
divide(10,2); // returns 5
divide(10,3); // returns 3.33333333

Recursion

```
Functions can call themselves  
long factorial (long n) {  
    if (n > 1)  
        return (n * factorial (n-1));  
    else  
        return (1);  
}
```

Prototyping

Functions can be prototyped so they can be used after being declared in any order
// prototyped functions can be used
// anywhere in the program
#include <iostream.h>
void odd (int a);
void even (int a);
int main () { ... }

Namespaces

Namespaces allow global identifiers under a name

```
// simple namespace  
namespace identifier {  
    namespace-body;  
}  
// example namespace  
namespace first {int var = 5;}  
namespace second {double var = 3.1416;}  
int main () {  
    cout << first::var << endl;  
    cout << second::var << endl;  
    return 0;  
}  
using namespace allows for the current nesting level to use the appropriate namespace  
using namespace identifier;  
// example using namespace  
namespace first {int var = 5;}  
namespace second {double var = 3.1416;}  
int main () {  
    using namespace second;  
    cout << var << endl;  
    cout << (var*2) << endl;  
    return 0;  
}
```

Class Reference

Class Syntax

```
class classname {
public:
    classname(params); // constructor
    ~classname();      // destructor
    member1;
    member2;
protected:
    member3;
    ...
private:
    member4;
} objectname;
// constructor (initializes variables)
classname::classname(params) {
}
// destructor (deletes variables)
classname::~classname() {
}

public members are accessible from anywhere
where the class is visible
protected members are only accessible from
members of the same class or of a friend class
private members are accessible from members
of the same class, members of the derived classes
and a friend class
constructors may be overloaded just like any
other function. define two identical constructors
with difference parameter lists

Class Example
class CSquare { // class declaration
public:
    void Init(float h, float w);
    float GetArea(); // functions
private: // available only to CSquare
    float h,w;
} // implementations of functions
void CSquare::Init(float hi, float wi){
    h = hi; w = wi;
}

float CSquare::GetArea() {
    return (h*w);
}

// example declaration and usage
CSquare theSquare;
theSquare.Init(8,5);
area = theSquare.GetArea();
// or using a pointer to the class
CSquare *theSquare;
theSquare->Init(8,5);
area = theSquare->GetArea();
```

Overloading Operators

Like functions, operators can be overloaded. Imagine you have a class that defines a square and you create two instances of the class. You can add the two objects together.

```
class CSquare { // declare a class
public: // functions
    void Init(float h, float w);
    float GetArea();
    CSquare operator + (CSquare);
private: // overload the '+' operator
    float h,w;
} // function implementations
void CSquare::Init(float hi, float wi){
    h = hi; w = wi;
}

float CSquare::GetArea() {
    return (h*w);
}

// implementation of overloaded operator
CSquare CSquare::operator+ (CSquare cs) {
    CSquare temp; // create CSquare object
    temp.h = h + cs.h; // add h and w to
    temp.w = w + cs.w; // temp object
    return (temp);
}

// object declaration and usage
CSquare sqr1, sqr2, sqr3;
sqr1.Init(3,4); // initialize objects
sqr2.Init(2,3);
sqr3 = sqr1 + sqr2; // object sqr3 is now
(5,7)
```

Advanced Class Syntax

Static Keyword
static variables are the same throughout all instances of a class.
static int n; // declaration
CDummy:n; // reference

Virtual Members
Classes may have virtual members. If the function is redefined in an inherited class, the parent must have the word **virtual** in front of the function definition

This keyword
The **this** keyword refers to the memory location of the current object.
int func(this); // passes pointer to
// current object

Class TypeCasting
reinterpret_cast <newtype>(expression);
dynamic_cast <newtype>(expression);
static_cast <newtype>(expression);
const_cast <newtype>(expression);

Expression Type
The type of an expression can be found using **typeid**. **typeid** returns a type.
typeid(expression);

Inheritance

Functions from a class can be inherited and reused in other classes. **Multiple inheritance** is possible.

```
class CPoly { //create base polygon class
protected:
    int width, height;
public:
    void SetValues(int a, int b)
    { width=a; height=b; }
};

class COutput { // create base output
public: // class
    void Output(int i);
};

void COutput::Output (int i) {
    cout << i << endl;
};

// CRect inherits SetValues from CPoly
// and inherits Output from COutput
class CRect: public CPoly, public COutput
{
public:
    int area(void)
    { return (width * height); }
};

// CTri inherits SetValues from CPoly
// and inherits Output from COutput
class CTri: public CPoly {
public:
    int area(void)
    { return (width * height / 2); }
};

void main () {
    CRect rect; // declare objects
    CRect tri;
    rect.SetValues (2,9);
    tri.SetValues (2,9);
    rect.Output(rect.area());
    cout<<tri.area()<<endl;
}
```

Templates

Templates allow functions and classes to be reused without overloading them

```
template <class id> function;
template <typename id> function;
// ----- function example -----
template <class T>
T GetMax (T a, T b) {
    return (a>b?a:b); // return the larger
}

void main () {
    int a=9, b=2, c;
    float x=5.3, y=3.2, z;
    c=GetMax(a,b);
    z=GetMax(x,y);
}

// ----- class example -----
template <class T>
class Pair {
    T x,y;
public:
    Pair(T a, T b){
        x=a; y=b; }
    T GetMax();
};

template <class T>
T Pair<T>::GetMax()
{ // implementation of GetMax function
    T ret; // return a template
    ret = x>y?x:y; // return larger
    return ret;
}

int main () {
    Pair <int> theMax (80, 45);
    cout << theMax.GetMax();
    return 0;
}
```

Friend Classes/Functions

Friend Class Example

```
class CSquare; // define CSquare
class CRectangle {
    int width, height;
public:
    void convert (CSquare a);
};

class CSquare { // we want to use the
private: // convert function in
    int side; // the CSquare class, so
public: // use the friend keyword
    void set_side (int a) { side=a; }
    friend class CRectangle;
};

void CRectangle::convert (CSquare a) {
    width = a.side;
    height = a.side;
}

// declaration and usage
CSquare sqr;
CRectangle rect; // convert can be
sqr.set_side(4); // used by the
rect.convert(sqr); // rectangle class

Friend Functions
A friend function has the keyword friend in front of it. If it is declared inside a class, that function can be called without reference from an object. An object may be passed to it.
/* change can be used anywhere and can
have a CRect object passed in */
// this example defined inside a class
friend CRect change(CRect);
CRectangle recta, rectb; // declaration
rectb = change(recta); // usage
```

File I/O

```
#include <fstream.h> // read/write file
#include <ofstream.h> // write file
#include <ifstream.h> // read file
File I/O is done from the fstream, ofstream, and
ifstream classes.
```

File Handles
A file must have a file handle (pointer to the file) to access the file.
ifstream infile; // create handle called
// infile to read from a file
ofstream outfile; // handle for writing
fstream f; // handle for read/write

Opening Files
After declaring a file handle, the following syntax can be used to open the file
void open(const char *fname, ios::mode);
fname should be a string, specifying an absolute or relative path, including filename. *ios::mode* can be any number of the following and repeat:
in Open file for reading
out Open file for writing
ate Initial position: end of file
app Every output is appended at the end of file
trunc If the file already existed it is erased
binary Binary mode
ifstream f; // open input file example
f.open("input.txt", ios::in);
ofstream f; // open for writing in binary
f.open("out.txt", ios::out | ios::binary
| ios::app);

Closing a File
A file can be closed by calling the handle's close function
f.close();

Writing To a File (Text Mode)
The operator **<<** can be used to write to a file. Like **cout**, a stream can be opened to a device. For file writing, the device is not the console, it is the file. **<<** is replaced with the file handle.
ofstream f; // create file handle
f.open("output.txt") // open file
f << "Hello World\n" << a < b < c < c < endl;

Reading From a File (Text Mode)
The operator **>>** can be used to read from a file. It works similar to **cin**. Fields are separated in the file by spaces.
ifstream f; // create file handle
f.open("input.txt") // open file
while (!f.eof()) // end of file test
f >> a >> b >> c; // read into a,b,c

I/O State Flags
Flags are set if errors or other conditions occur. The following functions are members of the file object
handle.bad() returns true if a failure occurs in reading or writing
handle.fail() returns true for same cases as **bad()** plus if formatting errors occur
handle.eof() returns true if the end of the file reached when reading
handle.good() returns false if any of the above were true

Stream Pointers
handle.tellg() returns pointer to current location when reading a file
handle.tellp() returns pointer to current location when writing a file
// seek a position in reading a file
handle.seekg(position);
handle.seekg(offset, direction);
// seek a position in writing a file
handle.seekp(position);
handle.seekp(offset, direction);
direction can be one of the following
ios::beg beginning of the stream
ios::cur current position of the stream pointer
ios::end end of the stream

Binary Files
buffer is a location to store the characters.
numbytes is the number of bytes to written or read.
write(char *buffer, numbytes);
read(char *buffer, numbytes);

Output Formatting
streamclass f; // declare file handle
// set output flags
f.flags(ios_base::flag)
possible flags
dec fixed hex oct
scientific internal left right
uppercase boolalpha showbase showpoint
showpos skips unitbuf
adjustfield left | right | internal
basefield dec | oct | hex
floatfield scientific | fixed
f.fill() get fill character
f.fill(ch) set fill character *ch*
f.precision(numdigits) sets the precision for floating point numbers to *numdigits*
f.put(c) put a single char into output stream
f.setf(flag) sets a flag
f.setf(flag, mask) sets a flag w/value
f.width() returns the current number of characters to be written
f.width(num) sets the number of chars to be written

ASCII Chart

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL	64	@	128	À	192	Ā
1	SOH	65	A	129	Á	193	Ă
2	STX	66	B	130	Â	194	Ą
3	ETX	67	C	131	Ã	195	Ć
4	EOT	68	D	132	Ä	196	Č
5	ENQ	69	E	133	Å	197	Ď
6	ACK	70	F	134	Æ	198	Đ
7	BEL	71	G	135	Ç	199	Ě
8	BS	72	H	136	Ĉ	200	Ħ
9	TAB	73	I	137	É	201	Į
10	LF	74	J	138	Ê	202	Ĳ
11	VTB	75	K	139	Ë	203	Ĵ
12	FF	76	L	140	Ė	204	Š
13	CR	77	M	141	Į	205	Ť
14	SO	78	N	142	Ā	206	Ț
15	SI	79	O	143	Ą	207	Ț
16	DLE	80	P	144	Ė	208	Ț
17	DC1	81	Q	145	æ	209	Ț
18	DC2	82	R	146	Æ	210	Ț
19	DC3	83	S	147	ö	211	Ț
20	DC4	84	T	148	ö	212	Ț
21	NAK	85	U	149	ö	213	Ț
22	SYN	86	V	150	ü	214	Ț
23	ETB	87	W	151	ü	215	Ț
24	CAN	88	X	152	ý	216	Ț
25	EM	89	Y	153	ÿ	217	Ț
26	SUB	90	Z	154	ÿ	218	Ț
27	ESC	91	[	155	ē	219	■
28	FS	92	\	156	ē	220	■
29	GS	93	]	157	£	221	■
30	RS	94	^	158	ƒ	222	?
31	US	95	_	159	ƒ	223	?
32	"	96	`	160	ā	224	ā
33	!	97	a	161	â	225	â
34	#"	98	b	162	ā	226	ā
35	#	99	c	163	ä	227	ä
36	\$	100	d	164	ā	228	ā
37	%	101	e	165	ñ	229	ñ
38	&	102	f	166	ā	230	μ
39	'	103	g	167	°	231	τ
40	(	104	h	168	ı	232	φ
41	)	105	i	169	ı	233	θ
42	*	106	j	170	ı	234	∞
43	+	107	k	171	½	235	δ
44	,	108	l	172	¼	236	∞
45	-	109	m	173	ı	237	ε
46	.	110	n	174	«	238	ε
47	/	111	o	175	»	239	∩
48	0	112	p	176	ı	240	≡
49	1	113	q	177	■	241	±
50	2	114	r	178	■	242	∞
51	3	115	s	179	ı	243	∞
52	4	116	t	180	ı	244	?
53	5	117	u	181	ı	245	?
54	6	118	v	182	ı	246	?
55	7	119	w	183	ı	247	?
56	8	120	x	184	ı	248	?
57	9	121	y	185	ı	249	?
58	:	122	z	186	ı	250	√
59	;	123	{	187	ı	251	√
60	<	124		188	ı	252	n
61	=	125	}	189	ı	253	z
62	>	126	~	190	ı	254	■
63	?	127	?	191	ı	255	ı

Dynamic Memory

Memory can be allocated and deallocated
// allocate memory (C++ only)
pointer = new type [i];
int *ptr; // declare a pointer
ptr = new int; // create a new instance
ptr = new int [5]; // new array of ints
// deallocate memory (C++ only)
delete [i] pointer;
delete ptr; // delete a single int
delete [i] ptr // delete array
// allocate memory (C or C++)
void * malloc (nbytes); // nbytes=size
char *buffer; // declare a buffer
// allocate 10 bytes to the buffer
buffer = (char *)malloc(10);
// allocate memory (C or C++)
buffer = (char *)malloc(10);
// nelements = number elements
// size = size of each element
void * malloc (nelements, size);
int *nums; // declare a buffer
// allocate 5 sets of ints
nums = (char *)calloc(5,sizeof(int));
// reallocate memory (C or C++)
void * realloc (*ptr, size);
// delete memory (C or C++)
void free (*ptr);

ANSI C++ Library Files

The following files are part of the ANSI C++ standard and should work in most compilers.
<algorithm.h> **<bitset.h>** **<deque.h>**
<exception.h> **<fstream.h>** **<functional.h>**
<iomanip.h> **<ios.h>** **<iosfwd.h>**
<iostream.h> **<iostream.h>** **<iterator.h>**
<limits.h> **<list.h>** **<locale.h>** **<map.h>**
<memory.h> **<new.h>** **<numeric.h>**
<ostream.h> **<queue.h>** **<set.h>** **<sstream.h>**
<stack.h> **<stdexcept.h>** **<streambuf.h>**
<string.h> **<typeinfo.h>** **<utility.h>**
<valarray.h> **<vector.h>**

C++ Reference Card

C/C++ Syntax, DataTypes, Functions
Classes, I/O Stream Library Functions

© 2002 The Book Company Storrs, CT

refcard@gbook.org

Information contained on this card carries no warranty. No liability is assumed by the maker of this card for accidents or damage resulting from its use.