KAIST
EE209: Programming Structures for EE
C Primitive Data Types

Type: int

Description: A (positive or negative) integer.

Size: System dependent. Usually either 2 or 4 bytes.

Example Variable Declarations:

```
int iFirst;
int iSecond, iThird;
signed int iFourth;
```

Example Literals (assuming size is 4 bytes):

C Literal	Binary Representation	Note
123	00000000 00000000 00000000 01111011	decimal form
-123	11111111 11111111 11111111 10000101	negative form
2147483647	01111111 11111111 11111111 11111111	largest
-2147483648	10000000 00000000 00000000 00000000	smallest
0173	00000000 00000000 00000000 01111011	octal form
0x7B	00000000 00000000 00000000 01111011	hexadecimal form

Type: unsigned int

Description: A non-negative integer.

Size: System dependent. Usually either 2 or 4 bytes. sizeof(unsigned int) == sizeof(int).

Example Variable Declarations:

```
unsigned int uiFirst;
unsigned int uiSecond, uiThird;
```

Example Literals (assuming size is 4 bytes):

C Literal	Binary Representation	Note
123U	00000000 00000000 00000000 01111011	decimal form
4294967295U	11111111 11111111 11111111 11111111	largest
0U	00000000 00000000 00000000 00000000	smallest
0173U	00000000 00000000 00000000 01111011	octal form
0x7BU	00000000 00000000 00000000 01111011	hexadecimal form

Type: long

Description: A (positive or negative) integer.

Size: System dependent. Usually 4 bytes. sizeof(long) >= sizeof(int).

Example Variable Declarations:

```
long lFirst;
long lSecond, lThird;
long int lFourth;
signed long lFifth;
signed long int lSixth;
```

Example Literals (assuming size is 4 bytes):

C Literal	Binary Representation	Note
123L	00000000 00000000 00000000 01111011	decimal form
-123L	11111111 11111111 11111111 10000101	negative form
2147483647L	01111111 11111111 11111111 11111111	largest
-2147483648L	10000000 00000000 00000000 00000000	smallest
0173L	00000000 00000000 00000000 01111011	octal form
0x7BL	00000000 00000000 00000000 01111011	hexadecimal form

Type: unsigned long

Description: A non-negative integer.

Size: System dependent. Usually 4 bytes. sizeof(unsigned long) == sizeof(long).

Example Variable Declarations:

```
unsigned long ulFirst;
unsigned long ulSecond, ulThird;
unsigned long int ulFourth;
```

Example Literals (assuming size is 4 bytes):

C Literal	Binary Representation	Note
123UL	00000000 00000000 00000000 01111011	decimal form
4294967295UL	11111111 11111111 11111111 11111111	largest
0UL	00000000 00000000 00000000 00000000	smallest
0173UL	00000000 00000000 00000000 01111011	octal form
0x7BUL	00000000 00000000 00000000 01111011	hexadecimal form

Type: char

Description: A (positive or negative) integer. Usually represents a character according to a character code (e.g., ASCII).

Size: 1 byte.

Example Variable Declarations:

```
char cFirst;
char cSecond, cThird;
signed char cFourth;
```

Example Literals (assuming the ASCII code is used):

C Literal	Binary Representation	Note
'a'	01100001	character form
(char)97	01100001	decimal form
(char)0141	01100001	octal form
(char)0x61	01100001	hexadecimal form
'\o141'	01100001	octal character form
'\x61'	01100001	hexadecimal character form
(char)123	01111011	decimal form
(char)-123	10000101	negative form
(char)127	01111111	largest
(char)-128	10000000	smallest
'\0'	00000000	the null character
'\a'	00000111	bell
'\b'	00001000	backspace
'\f'	00001100	formfeed
'\n'	00001010	newline
'\r'	00001101	carriage return
'\t'	00001001	horizontal tab
'\v'	00001011	vertical tab
'\\'	01011100	backslash
'\''	00100111	single quote

Type: unsigned char

Description: A non-negative integer. Usually represents a character according to a character code (e.g., ASCII).

Size: 1 byte.

Example Variable Declarations:

```
unsigned char ucFirst;
unsigned char ucSecond, ucThird;
```

Example Literals (assuming the ASCII code is used):

C Literal	Binary Representation	Note
(unsigned char)'a'	01100001	character form
(unsigned char)97	01100001	decimal form
(unsigned char)255	11111111	largest
(unsigned char)0	00000000	smallest

Note: On most systems, "char" is the same as "signed char".
On some systems, "char" is the same as "unsigned char".

Type: short

Description: A (positive or negative) integer.

Size: System dependent. Usually 2 bytes. sizeof(short) <= sizeof(int).

Example Variable Declarations:

```
short sFirst;
short sSecond, sThird;
short int sFourth;
signed short sFifth;
signed short int sSixth;
```

Example Literals (assuming size is 2 bytes):

C Literal	Binary Representation	Note
(short)123	00000000 01111011	decimal form
(short)-123	11111111 10000101	negative form
(short)32767	01111111 11111111	largest
(short)-32768	10000000 00000000	smallest
(short)0173	00000000 01111011	octal form
(short)0x7B	00000000 01111011	hexadecimal form

Type: unsigned short

Description: A non-negative integer.

Size: System dependent. Usually 2 bytes. sizeof(unsigned short) == sizeof(short).

Example Variable Declarations:

```
unsigned short usFirst;
unsigned short usSecond, usThird;
unsigned short int usFourth;
```

Example Literals (assuming size is 2 bytes):

C Literal	Binary Representation	Note
(unsigned short)123	00000000 01111011	decimal form

(unsigned short)65535	11111111 11111111	largest
(unsigned short)0	00000000 00000000	smallest
(unsigned short)0173	00000000 01111011	octal form
(unsigned short)0x7B	00000000 01111011	hexadecimal form

Type: double

Description: A (positive or negative) double-precision floating point number.

Size: System dependent. Often 8 bytes.

Example Variable Declarations:

```
double dFirst;
double dSecond, dThird;
```

Example Literals (assuming size is 8 bytes):

C Literal	Note
123.456	fixed-point notation
1.23456E2	scientific notation
.0123456	fixed-point notation
1.234546E-2	scientific notation with negative exponent
-123.456	fixed-point notation
-1.23456E2	scientific notation with negative mantissa
-.0123456	fixed-point notation
-1.23456E-2	scientific notation with negative mantissa and negative exponent
1.797693E308	largest (approximate)
-1.797693E308	smallest (approximate)
2.225074E-308	closest to 0 (approximate)

Type: float

Description: A (positive or negative) single-precision floating point number.

Size: System dependent. Often 4 bytes. sizeof(float) <= sizeof(double).

Example Variable Declarations:

```
float fFirst;
float fSecond, fThird;
```

Example Literals (assuming size is 4 bytes):

C Literal	Note
123.456F	fixed-point notation
1.23456E2F	scientific notation
.0123456F	fixed-point notation
1.234546E-2F	scientific notation with negative exponent
-123.456F	fixed-point notation
-1.23456E2F	scientific notation with negative mantissa
-.0123456F	fixed-point notation
-1.23456E-2F	scientific notation with negative mantissa and negative exponent
3.402823E38F	largest (approximate)
-3.402823E38F	smallest (approximate)
1.175494E-38F	closest to 0 (approximate)

Type: long double

Description: A (positive or negative) extended-precision floating point number.

Size: System dependent. Often 12 bytes. sizeof(long double) >= sizeof(double).

Example Variable Declarations:

```
long double ldFirst;
long double ldSecond, ldThird;
```

Example Literals (assuming size is 12 bytes):

C Literal	Note
123.456L	fixed-point notation
1.23456E2L	scientific notation
.0123456L	fixed-point notation
1.234546E-2L	scientific notation with negative exponent
-123.456L	fixed-point notation
-1.23456E2L	scientific notation with negative mantissa
-.0123456L	fixed-point notation
-1.23456E-2L	scientific notation with negative mantissa and negative exponent
1.189731E4932L	largest (approximate)
-1.189731E4932L	smallest (approximate)
3.362103E-4932L	closest to 0 (approximate)

Differences between C and Java:

Java only:
boolean, byte

C only:
unsigned char, unsigned short, unsigned int, unsigned long
long double

Java: Sizes of all types are **specified**
C: Sizes of all types except char are **system dependent**

Java: char comprises **2** bytes
C: char comprises **1** byte

KAIST
EE 209: Programming Structures for EE
C Symbolic Constants

Method 1: #define

Example

```
int main(void)
{
  #define START_STATE 0
  #define POSSIBLE_COMMENT_STATE 1
  #define COMMENT_STATE 2
  ...
  int iState;
  ...
  iState = START_STATE;
  ...
}
```

Strengths

Preprocessor does substitutions only for tokens.

```
int iSTART_STATE; /* No substitution. */
```

Preprocessor does not do substitutions within string constants.

```
printf("What is the START_STATE?\n"); /* No substitution. */
```

Simple textual substitution; works for any type of data.

```
#define PI 3.14159
```

Weaknesses

Preprocessor does not respect context.

```
int START_STATE;
```

After preprocessing, becomes:

```
int 0; /* Compiletime error. */
```

Convention: Use all uppercase letters to reduce probability of unintended replacement.

Preprocessor does not respect scope.

Preprocessor replaces START_STATE with 0 from point of #define to end of *file*, not to end of *function*. Could affect subsequent functions unintentionally.

Convention: Place #defines at beginning of file, not within function definitions

Method 2: Constant Variables

Example

```
int main(void)
{
  const int START_STATE = 0;
  const int POSSIBLE_COMMENT_STATE = 1;
  const int COMMENT_STATE = 2;
  ...
  ...
  int iState;
  ...
  iState = START_STATE;
  ...
  iState = COMMENT_STATE;
  ...
}
```

Strengths

Works for any type of data.

```
const double PI = 3.14159;
```

Handled by compiler; compiler respects context and scope.

Weaknesses

Does not work for array lengths (unlike C++).

```
const int ARRAY_LENGTH = 10;
...
int a[ARRAY_LENGTH]; /* Compiletime error */
```

Method 3: Enumerations

Example

```
int main(void)
{
  /* Define a type named "enum State". */
  enum State {START_STATE, POSSIBLE_COMMENT_STATE, COMMENT_STATE, ...};
  /* Declare "eState" to be a variable of type "enum State".
  enum State eState;
  ...
  eState = START_STATE;
  ...
  eState = COMMENT_STATE;
  ...
}
```

Notes

Interchangeable with type int.

```
eState = 0; /* Can assign int to enum. */

i = START_STATE; /* Can assign enum to int. START_STATE is an alias for
                   0  , POSSIBLE_COMMENT_STATE is an alias for 1, etc. */
```

Strengths

Can explicitly specify values for names.

```
enum State {START_STATE = 5,
            POSSIBLE_COMMENT_STATE = 3,
            COMMENT_STATE = 4,
            ...};
```

Can omit type name, thus effectively giving symbolic names to int literals.

```
enum {MAX_VALUE = 9999};
...
int i;
...
i = MAX_VALUE;
...
```

Works when specifying array lengths.

```
enum {ARRAY_LENGTH = 10};
...
int a[ARRAY_LENGTH];
...
```

Weakness

Does not work for non-integral data types.

```
enum {PI = 3.14159}; /* Compile-time error */
```

Style Rules (see Kernighan and Pike Chapter 1)

(1) Use **enumerations** to give symbolic names to **integral** literals.

(2) Use **const variables** to give symbolic names to **non-integral** literals.

(3) Avoid using **#define**.

KAIST
EE209: Programming Structures for EE
C Statements

Statement Type	Statement Syntax	Examples

Expression Statement	*expression*;	i = 5;
		printf("Hello");
		5; /* valid, but nonsensical */

Declaration Statement	*modifiers datatype variable [= initialvalue][,variable [= initialvalue]]...;*	int i; int i, j; int i = 5, j = 6; const int i; static int i; extern int i;

Compound Statement (alias Block)	{*statement statement* ... }	{
		    int i;
		    i = 5;
		...
		}

If Statement	if (*integralexpr*) *statement*;
	if (*pointerexpr*) *statement*;
		if (i == 5)
		{
		    statement;
		    statement;
		}

Switch Statement	switch (*integralexpr*)
	{
	case *integralconstant*: *statements*
	case *integralconstant*: *statements*
	default: *statements*
	}
		switch (i)
		{
		    case 1: *statement*; break;
		    case 2: *statement*; break;
		    default: *statement*;
		}

While Statement	while (*integralexpr*) *statement*	while (i < 5)
		{
		    statement;
		    statement;
		}

DoWhile Statement	do *statement* while (*integralexpr*);	do
		{
		    statement;
		    statement;
		} while (i < 5);

For Statement	for (*initexpr*; *integralexpr*; *increxpr*) *statement*	for (i = 0; i < 5; i++)
		{
		    statement;
		    statement;
		}

Return Statement	return; return *expr*;	return; return i + 5;

Break Statement	break;	while (i < 5)
		{
		    statement;
		    if (j == 6)
		    break;
		    statement;
		}

Continue Statement	continue;	

```
while (i < 5)
{
      statement;
      if (j == 6)
      continue;
      statement;
}
```

Goto Statement	goto *label*;	

```
mylabel:
      …
      goto mylabel;
      …
```

Differences between C and Java:

Expression Statement:

Java: Only expressions that have a side effect can be made into expression statements
C: Any expression can be made into an expression statement
Java: Has "final" variables
C: Has "const" variables

Declaration Statement:

Java: Compiletime error to use a local variable before specifying its value
C: Runtime error to use a local variable before specifying its value

Compound Statement:

Java: Declarations statements can be placed anywhere within compound statement
C: Declaration statements must appear before any other type of statement within compound statement

If Statement

Java: Controlling expr must be of type boolean
C: Controlling expr must be of some integral type or a pointer (0 => FALSE, non-0 => TRUE)

While Statement

Java: Controlling expr must be of type boolean
C: Controlling expr must be of some integral type or a pointer (0 => FALSE, non-0 => TRUE)

DoWhile Statement

Java: Controlling expr must be of type boolean
C: Controlling expr must be of some integral type or a pointer (0 => FALSE, non-0 => TRUE)

For Statement

Java: Controlling expr must be of type boolean
C: Controlling expr must be of some integral type or a pointer (0 => FALSE, non-0 => TRUE)
Java: Can declare loop control variable in initexpr
C: Cannot declare loop control variable in initexpr

Break Statement

Java: Also has "labeled break" statement
C: Does not have "labeled break" statement

Continue Statement

Java: Also has ″labeled continue″ statement

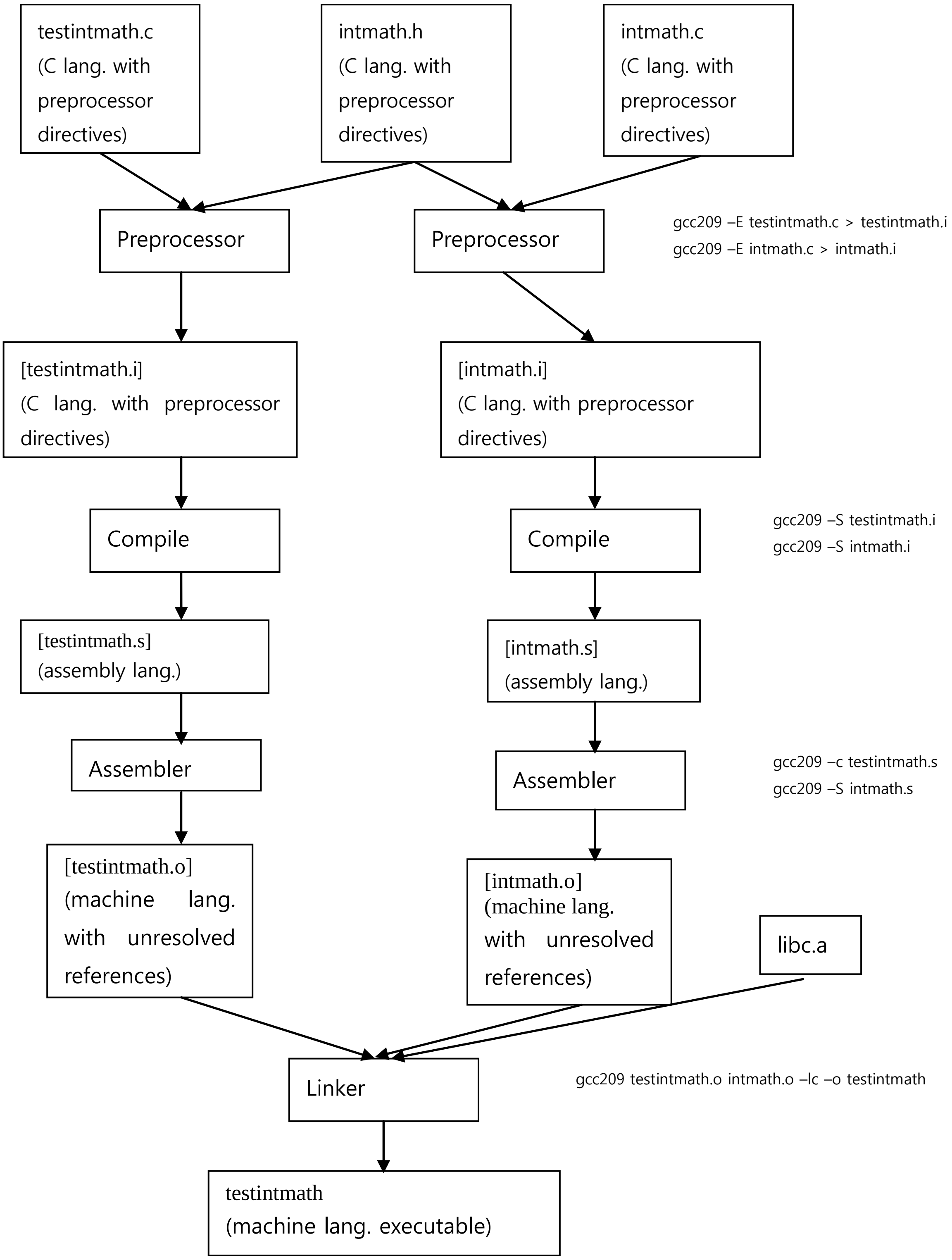
C: Does not have ″labeled continue″ statement

Goto Statement

Java: Not provided

C: Provided (but don't use it!)

KAIST
EE209: Programming Structures for EE
Building Multi-File C Programs



```
gcc209 –E testintmath.c > testintmath.i
gcc209 –E intmath.c > intmath.i
```

```
gcc209 –S testintmath.i
gcc209 –S intmath.i
```

```
gcc209 –c testintmath.s
gcc209 –S intmath.s
```

```
gcc209 testintmath.o intmath.o –lc –o testintmath
```

Shortcut:

```
gcc209 testintmath.c intmath.c –o testintmath
```

KAIST
EE209: Programming Structures for EE
GDB Tutorial

This tutorial describes how to use a minimal subset of the GDB debugger. See the summary sheet distributed in precept for more information. Also see Chapter 6 of our *Programming with GNU Software* (Loukides & Oram) textbook.

The tutorial assumes that you have created files named testintmath.c, intmath.h, and intmath.c in your working directory, containing the (version 4) program recently discussed in precepts. Those files are available through the course "Schedule" Web page.

Introduction

Suppose you are developing the testintmath (version 4) program. Further suppose that the program preprocesses, compiles, assembles, and links cleanly, but is producing incorrect results at runtime. What can you do to debug the program?

One approach is temporarily to insert calls to printf(...) or fprintf(stderr, ...) throughout the code to get a sense of the flow of control and the values of variables at critical points. That's fine, but often is inconvenient.

An alternative is to use GDB. GDB is a powerful debugger. It allows you to set breakpoints in your code, step through your executing program one line at a time, examine the values of variables at breakpoints, examine the function call stack, etc.

Building for GDB

To prepare to use GDB, build your program with the -g option:

```
$ gcc209 -g testintmath.c intmath.c -o testintmath
```

Doing so places extra information into the testintmath file that GDB uses.

Running GDB

The next step is to run GDB. You can run GDB directly from the shell, but it's much handier to run it from within Emacs. So launch Emacs, with no command-line arguments:

```
$ emacs
```

Now call the Emacs "gdb" function via these keystrokes:

```
<Esc key> x gdb <Enter key> testintmath <Enter key>
```

At this point you are executing GDB from within Emacs. GDB is displaying its (gdb) prompt.

Running your Program

Issue the "run" command to run the program:

```
(gdb) run
```

Enter 8 as the first integer, and 12 as the second integer. GDB runs the program to completion, indicating that the "Program exited normally." Incidentally, file redirection is specified as part of the "run" command. For example, the command "run < *somefile*" runs the program, redirecting standard input to *somefile*.

Using Breakpoints

Set a breakpoint at the beginnings of some functions using the "break" command:

```
(gdb) break main
(gdb) break IntMath_gcd
```

(Incidentally, another way to set a breakpoint is by specifying a file name and line number separated by a colon, for example, "break intmath.c:20".) Run the program:

```
(gdb) run
```

GDB pauses execution near the beginning of main(). It opens a second window in which it displays your source code, with the about-to-be-executed line of code highlighted.

Issue the "continue" command to tell command GDB to continue execution past the breakpoint:

```
(gdb) continue
```

GDB continues past the breakpoint at the beginning of main(), and execution is paused at a scanf(). Enter 8 as the first number. Execution is paused at the second scanf(). Enter 12 as the second number. GDB is paused at the beginning of IntMath_gcd().

Then issue another "continue" command:

```
(gdb) continue
```

Note that GDB is paused, again, at the beginning of IntMath_gcd(). (Recall the IntMath_gcd() is called twice: once by main(), and once by IntMath_lcm().)

While paused at a breakpoint, issue the "kill" command to stop execution:

```
(gdb) kill
```

Type "y" to confirm that you want GDB to stop execution.

Issue the "clear" command to get rid of a breakpoint:

```
(gdb) clear IntMath_gcd
```

At this point only one breakpoint remains: the one at the beginning of main().

Stepping through the Program

Run the program again:

```
(gdb) run
```

Execution pauses at the beginning of main(). Issue the "next" command to execute the next line of your program:

```
(gdb) next
```

Continue issuing the "next" command repeatedly until the program ends.
Run the program again:

```
(gdb) run
```

Execution pauses at the beginning of main(). Issue the "step" command to execute the next line of your program:

```
(gdb) step
```

Continue issuing the "step" command repeatedly until the program ends. Is the difference between "next" and "step" clear? The "next" command tells GDB to execute the next line, while staying at the same function call level. In contrast, the "step" command tells GDB to step into a called function.

Examining Variables

Set a breakpoint at the beginning of IntMath_gcd():

```
(gdb) break IntMath_gcd
```

Run the program until execution reaches that breakpoint:

```
(gdb) run
(gdb) continue
```

Now issue the "print" command to examine the values of the parameters of IntMath_gcd():

```
(gdb) print iFirst
```

```
(gdb) print iSecond
```

In general, when paused at a breakpoint you can issue the "print" command to examine the value of any expression containing variables that are in scope.

Examining the Call Stack

While paused at IntMath_gcd(), issue the "where" command:

```
(gdb) where
```

In response, GDB displays a call stack trace. Reading the output from bottom to top gives you a trace from a specific line of the main() function, through specific lines of intermediate functions, to the about-to-be-executed line.

The "where" command is particularly useful when your program is crashing via a "segmentation fault" error at runtime. When that occurs, try to make the error occur within GDB. Then, after the program has crashed, issue the "where" command. Doing so will give you a good idea of which line of your code is causing the error.

Quitting GDB

Issue the "quit" command to quit GDB:

```
(gdb) quit
```

Then, as usual, type:

```
<Ctrl-x> <Ctrl-c>
```

to exit Emacs.

Command Abbreviations

The most commonly used GDB commands have one-letter abbreviations (r, b, c, n, s, p). Also, pressing the Enter key without typing a command tells GDB to reissue the previous command.

KAIST
EE209: Programming Structures for EE
The GDB Debugger for C Programs

gcc209 –g … -o program	Build with debugging information
gdb [-d sourcefiledir] [-d sourcefiledir] … program [corefile]	Run GDB from a shell
ESC x gdb gdb [-d sourcefiledir] [-d sourcefiledir] … program [corefile]	Run GDB within Emacs

Miscellaneous	
quit	Exit GDB.
directory [*dir1*] [*dir2*] ...	Add directories *dir1*, *dir2*, ... to the list of directories searched for source files, or clear the directory list.
help [*cmd*]	Print a description of command *cmd*.

Listing the Source Code (or run within Emacs)	
list [[*file*:]*linenum1*[-*linenum2*]]	Print the source code lines numbered *linenum1* to *linenum2* in file *file*.
list [[*file*:]*fn*:][*linenum1*[-*linenum2*]]	Print the source code lines numbered *linenum1* to *linenum2* in function *fn* in file *file*.

Running the Program	
run [*arg1*],[*arg2*] ...	Run the program with command-line arguments *arg1*, *arg2*, ...
set args *arg1 arg2 ...*	Set the program's command-line arguments to *arg1*, *arg2*, ...
show args	Print the program's command-line arguments.

Using Breakpoints	
info breakpoints	Print a list of all breakpoints.
break [*file*:]*linenum*	Set a breakpoint at line *linenum* in file *file*.
break [*file*:]*fn*	Set a breakpoint at the beginning of function *fn* in file *file*.
condition *bpnum expr*	Break at breakpoint *bpnum* only if expression *expr* is non-zero (TRUE).
commands [*bpnum*] *cmds*	Execute commands *cmds* whenever breakpoint *bpnum* is hit.
continue	Continue executing the program.
kill	Stop executing the program.
delete [*bpnum1*][,*bpnum2*]...	Delete breakpoints *bpnum1*, *bpnum2*, ..., or all breakpoints.
clear [[*file*:]*linenum*]	Clear the breakpoint at *linenum* in file *file*, or the current breakpoint.
clear [[*file*:]*fn*]	Clear the breakpoint at the beginning of function *fn* in file *file*, or the current breakpoint.
disable [*bpnum1*][,*bpnum2*]...	Disable breakpoints *bpnum1*, *bpnum2*, ..., or all breakpoints.
enable [*bpnum1*][,*bpnum2*]...	Enable breakpoints *bpnum1*, *bpnum2*, ..., or all breakpoints.

Stepping through the Program	
next	"Step over" the next line of the program.
step	"Step into" the next line of the program.
finish	"Step out" of the current function.

Examining Variables	

print *expr*	Print the value of expression *expr*.
print ['*file*'::]*var*	Print the value of variable *var* as defined in file *file*. (*File* is used to resolve static variables.)
print [*function*::]*var*	Print the value of variable *var* as defined in function *function*. (*Function* is used to resolve static variables.)
printf *format*, *expr1*, *expr2*, ...	Print the values expressions *expr1*, *expr2*, ... using the specified *format* string.
whatis *var*	Print the type of variable *var*.
ptype *t*	Print the definition of type *t*.
info display	Print the display list.
display *expr*	At each break, print the value of expression *expr*.
undisplay *displaynum*	Remove *displaynum* from the display list.

Examining the Call Stack	
where	Print the call stack.
backtrace	Print the call stack.
frame	Print the top of the call stack.
up	Move the context toward the bottom of the call stack.
down	Move the context toward the top of the call stack.

Working with Signals	
info signals	Print a list of all signals that the operating system makes available.
handle *sig action1* [*action2* ...]	When GDB receives signal *sig*, it should perform actions *action1*, *action2*, ... Valid actions are nostop, stop, print, noprint, pass, and nopass.
signal *sig*	Send the program signal *sig*.