CSC3002 Project Report

Jiayin Liu 115010048 Xinyang Tan 117010242 Hanzhi Chen 118010011
Jiaqi Chen 118010014 Jianing Liu 118010182 Yudi Yang 118010373

Introduction:

We have designed an IDE (Integrated Development Environment) in the past few months, which uses the "C ++ language" style as syntax. Although the function of our IDE is simplified compared with the popular IDE, such as Qt, Visual Studio, it has a comprehensive structure and architecture for people to manage programming from inputting and visualizing the source codes to getting the output as the result of execution. The general structure of our IDE is shown below (**Figure 1**):

• **Editor:**

A source code editor is a text editor program designed specifically for editing source code of computer programs which is built into an integrated development environment (IDE).

• **Compiler:**

A compiler is used for turning the source code into an assembly language. Particularly, our compiler can realize the function to transform part of the source code into MIPS.

• **Assembler:**

Assembler is a computer program that reads source code written in assembly language and produces executable machine code. Our assembler turns the MIPS generated by the compiler into the machine-executable application

• **Linker**

Linker intakes the object codes generated by the assembler and combines them to generate the executable module.

• **Loader**

The loader loads this executable module to the main memory for execution. In our program, the loader has been combined with a simulator together.

• **Simulator:**

The simulator operates instructions from the generated machine code.

• **Debugger:**

A debugger is a computer program used to test and debug other programs. It permits the programmer to track its operations in progress and monitor changes in computer resources that may indicate malfunctioning code.

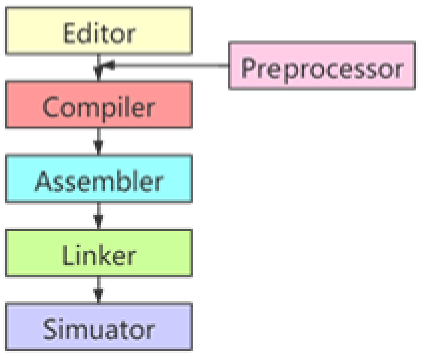


Figure 1

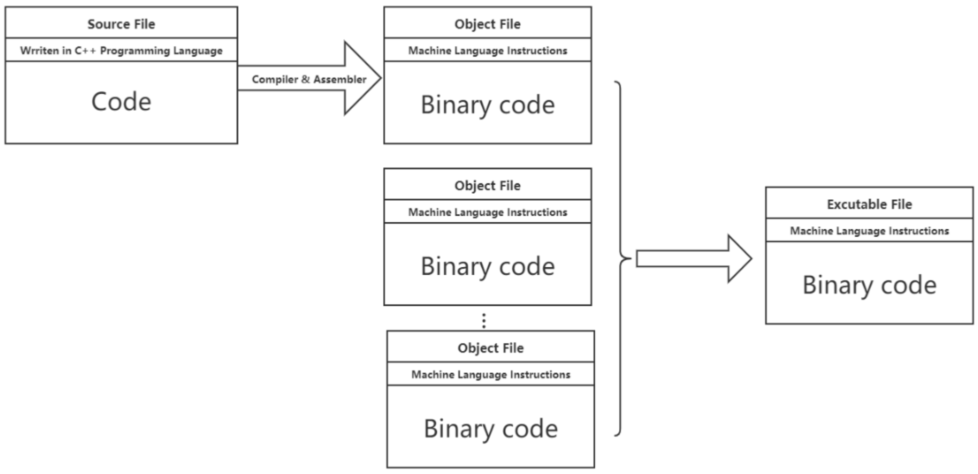
Throughout the project, we completed a short version of the IDE through team members' remote meetings and cooperation. This is one of the few team programming experiences we have in universities, and it is also the first time we have used C ++ to complete a team programming project. Although we encountered many challenges throughout the process, including the difficulties that will be encountered in offline cooperation and the difficulties of remote communication and collaboration that could not be encountered in offline cooperation, each of us makes every effort to complete the work and tasks of the group project.

This project provided us a holistic understanding of how computer run codes and helped us build upon the skills in design software with C++ on scratch. In addition, it can also facilitate our soft skills, such as communication skills and cooperation skills, which is useful for our future career.

Literature Review

We got a fundamental understanding of the architecture and functions of IDE from the book, *Programming Abstractions in C++*. As mentioned in the book, when clients write programs in C++, what they first need to do is to create a **source file** that contains the text of the program.

Then the IDE will translate the source file into an object file with the corresponding machine language instructions using **compiler & assembler**. This object file will combine with other object files to be an executable file by **linkers**. Libraries which are collections of previously written tools that perform useful operations are essentially a kind of object file. The process of transforming the program file is shown in **Figure 2**:



Source File
Wrriten in C++ Programming Language
Code
Compiler & Assembler
Object File
Machine Language Instructions
Binary code
Object File
Machine Language Instructions
Binary code
Object File
Machine Language Instructions
Binary code
Excutable File
Machine Language Instructions
Binary code

Figure 2

The main part of the editor is a GUI(Graphical User Interface) in which a person can communicate with a computer. The book, *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques*, describes how to design a GUI. We need to design icons or menu options on the screen for users to manipulate using input devices such as a mouse to select commands, call files, start programs, or perform other daily tasks. Compared to command-line interfaces that use keyboards to enter text or character commands to complete routine tasks, GUI has many advantages. The graphical user interface is composed of windows, drop-down menus, dialog boxes and their corresponding control mechanisms. It helps us visualize what we have input and results as well as what we are typing while it can also standardize the operation from the clients, that is, the same operation is always done in the same way. The GUI designs we mainly refer to are from a text editor **(Figure 3)**, UltraEdit **(Figure 4)**, and QT Creator **(Figure 5)**.

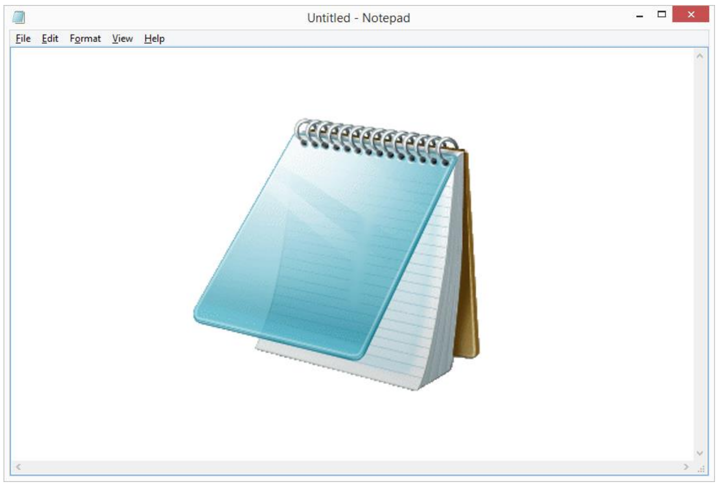
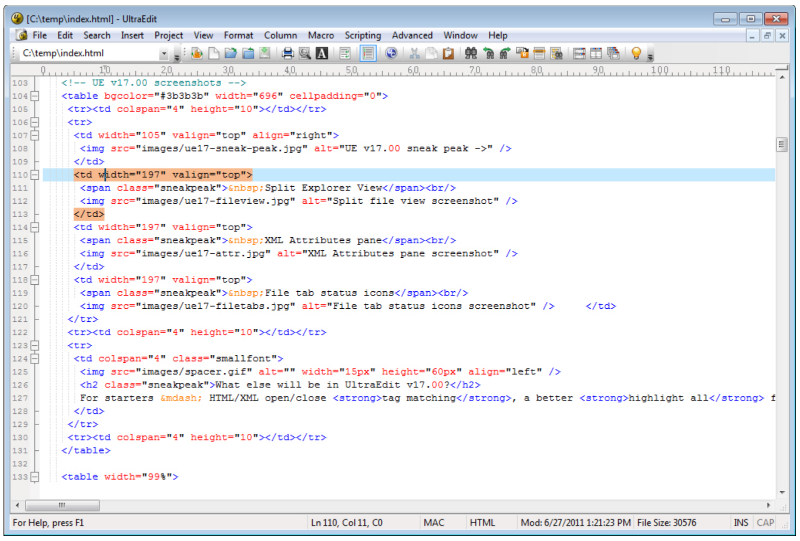


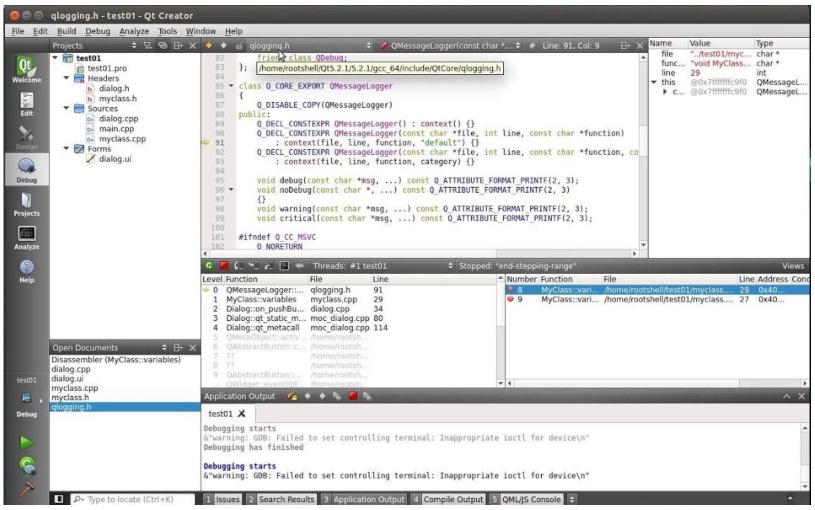
Figure 3


[C:\temp\index.html] - UltraEdit
File Edit Search Insert Project View Format Column Macro Scripting Advanced Window Help
C:\temp\index.html

```
<!-- UE v17.00 screenshots -->
<table bgcolor="#3b3b3b" width="696" cellpadding="0">
 <tr><td colspan="4" height="10"></td></tr>
 <tr>
  <td width="105" valign="top" align="right">
   <img src="images/ue17-sneak-peak.jpg" alt="UE v17.00 sneak peak ->" />
  </td>
  <td width="197" valign="top">
   <span class="sneakpeak">&nbsp;Split Explorer View</span><br/>
   <img src="images/ue17-fileview.jpg" alt="Split file view screenshot" />
  </td>
  <td width="197" valign="top">
   <span class="sneakpeak">&nbsp;XML Attributes pane</span><br/>
   <img src="images/ue17-attr.jpg" alt="XML Attributes pane screenshot" />
  </td>
  <td width="197" valign="top">
   <span class="sneakpeak">&nbsp;File tab status icons</span><br/>
   <img src="images/ue17-filetabs.jpg" alt="File tab status icons screenshot" />     </td>
 </tr>
 <tr><td colspan="4" height="10"></td></tr>
 <tr>
  <td colspan="4" class="smallfont">
   <img src="images/spacer.gif" alt="" width="15px" height="60px" align="left" />
   <h2 class="sneakpeak">What else will be in UltraEdit v17.00?</h2>
   For starters &mdash; HTML/XML open/close <strong>tag matching</strong>, a better <strong>highlight all</strong> f
  </td>
 </tr>
 <tr><td colspan="4" height="10"></td></tr>
</table>

<table width="99%">
```

```
For Help, press F1    Ln 110, Col 11, C0    MAC    HTML    Mod: 6/27/2011 1:21:23 PM    File Size: 30576    INS    CAP
```

Figure 4



```
qlogging.h    QMessageLogger(const char *,...    Line: 91, Col: 9
/home/rootshell/Qt5.2.1/5.2.1/gcc_64/include/QtCore/qlogging.h

    friend class QDebug;
};

class Q_CORE_EXPORT QMessageLogger
{
    Q_DISABLE_COPY(QMessageLogger)
public:
    Q_DECL_CONSTEXPR QMessageLogger() : context() {}
    Q_DECL_CONSTEXPR QMessageLogger(const char *file, int line, const char *function)
        : context(file, line, function, "default") {}
    Q_DECL_CONSTEXPR QMessageLogger(const char *file, int line, const char *function, co
        : context(file, line, function, category) {}

    void debug(const char *msg, ...) const Q_ATTRIBUTE_FORMAT_PRINTF(2, 3);
    void noDebug(const char *, ...) const Q_ATTRIBUTE_FORMAT_PRINTF(2, 3)
    {}
    void warning(const char *msg, ...) const Q_ATTRIBUTE_FORMAT_PRINTF(2, 3);
    void critical(const char *msg, ...) const Q_ATTRIBUTE_FORMAT_PRINTF(2, 3);

#ifndef Q_CC_MSVC
    Q_NORETURN
```

```
Name     Value                  Type
file     "../test01/myc...      char *
func...  "void MyClass...       char *
line     29                     int
this     @0x7fffffffc9f0        QMessageL...
 c...    @0x7fffffffc9f0        QMessageL...
```

```
Threads: #1 test01    Stopped: "end-stepping-range"

Level Function               File             Line
 0    QMessageLogger::...    qlogging.h       91
 1    MyClass::variables     myclass.cpp      29
 2    Dialog::on_pushBu...   dialog.cpp       34
 3    Dialog::qt_static_m... moc_dialog.cpp   80
 4    Dialog::qt_metacall    moc_dialog.cpp   114
 5    QMetaObject::activ...  /home/rootsh...
 6    QAbstractButton::c...  /home/rootsh...
 7    ??                     /home/rootsh...
 8    ??                     /home/rootsh...
 9    QAbstractButton::...   /home/rootsh...
 ...  QWidget::event(QE...   /home/rootsh...

Number Function          File                                  Line Address Cond
 8     MyClass::vari...  /home/rootshell/test01/myclass....    29   0x40...
 9     MyClass::vari...  /home/rootshell/test01/myclass....    27   0x40...
```

```
Application Output
test01
Debugging starts
&"warning: GDB: Failed to set controlling terminal: Inappropriate ioctl for device\n"
Debugging has finished

Debugging starts
&"warning: GDB: Failed to set controlling terminal: Inappropriate ioctl for device\n"
```

Figure 5

Compiler, a software translator, usually accepts program written in high-level language as input and produces an equivalent program in machine language as output*(Compiler Design: Theory, Tools, and Examples C/C++ Edition Seth D. Bergmann, Rowan University, 2010)*. The main workflow of a modern compiler can be roughly described as the following **(Figure 6)**:

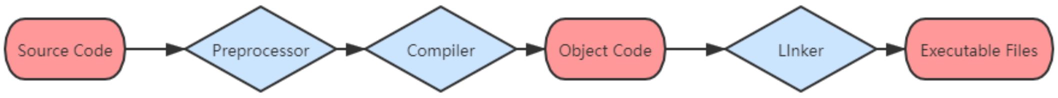


Figure 6

The compiler converts the entire program into an object code that is usually stored in a file. The target code is also called binary code and can be directly executed by the machine after linking **(Figure 7)**. Interpreter directly executes instructions written in a programming or scripting language, analyzes and executes each line of source code in succession, without looking at the entire program. Usually, the program generated by the compiler runs much faster than the same program executed by the interpreter.

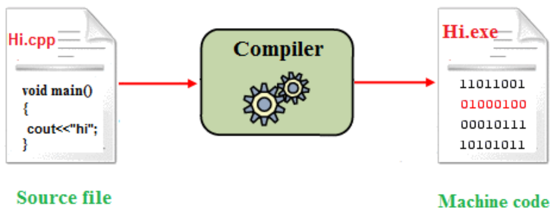


Figure 7

The assembler converts assembly language statement files into binary machine instructions and binary data files **(Figure 8)**. The translation process is divided into two main parts. The first step is to find the storage location with the label so that you can know the relationship between the symbol name and the address when translating the instruction. The second step is to convert each assembly statement by combining the numeric equivalents of the opcode, registration specifier and label into legal instructions.

Assembly language has a lot of shortcomings. First, the program written by assembly language is inherently machine-specific. Second, it is longer than the equivalent programs written in high-level language. However, there is no denying that it has been widely used by people in recent years.

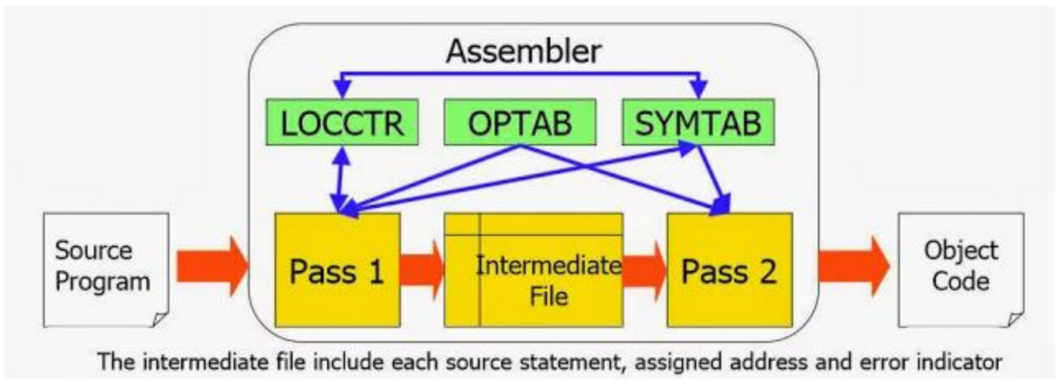

Assembler
LOCCTR
OPTAB
SYMTAB
Source Program
Pass 1
Intermediate File
Pass 2
Object Code
The intermediate file include each source statement, assigned address and error indicator

Figure 8

The simulator runs assembly language programs written for processors that usually implement the MIPS-32 architecture. PIC Simulator IDE supplies Microchip microcontroller users with user-friendly graphical development environment for Windows with integrated simulator (emulator), pic basic compiler, assembler, disassembler and debugger.

Separate compilation allows the program to be split into multiple fragments and stored in separate files. Separate compilation requires linking to the additional step of merging object files from separate modules and fixing their unresolved references. The tool to merge these files is the linker **(Figure 9)**. Linker mainly performs three tasks:

■ Search the library to find the library routine used by the program

■ Determine the memory location that each module's code will occupy, and relocate its instructions by adjusting the absolute reference

■ Resolve references between files

The primary task of the linker is to ensure that the program does not contain undefined tags. The linker will match external symbols and unresolved references in program files. If both files refer to tags with the same name, the external symbol in one file will resolve the reference from the other file. Reference mismatch means that the symbol is used anywhere in the program but the symbol is not defined.

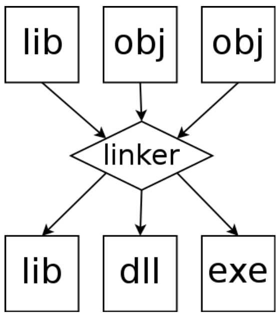


Figure 9

Debugging tools are also called debugging programs and debuggers, which refer to a computer program and tool for debugging other programs. It enables the code to check the running status and selectively run in the instruction group simulator for debugging. This technique can be very useful when the development progress encounters a bottleneck or cannot find implicit problems. However, running the program under the debugger is usually slower than running directly on the operating platform and processors.

Our Work

◆ **Editor:**

In this project, we designed an editor with an awesome graphical user interface as shown below:

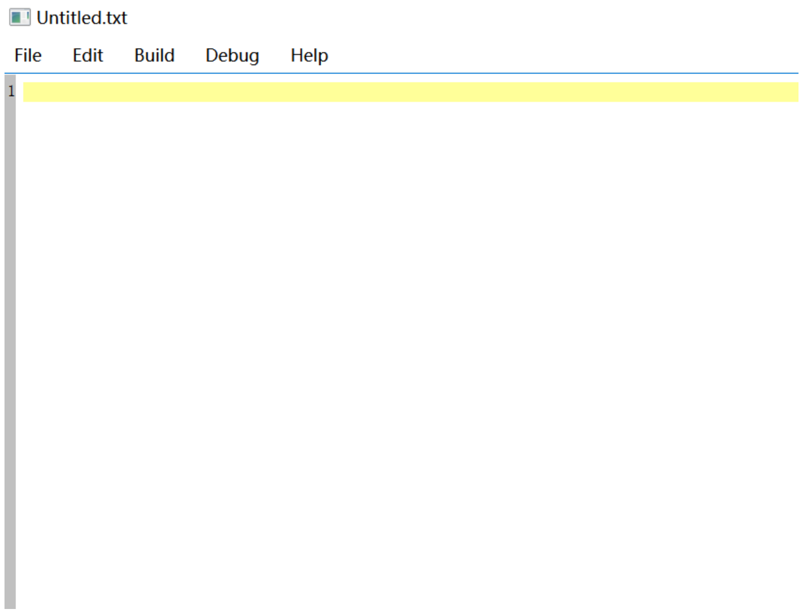
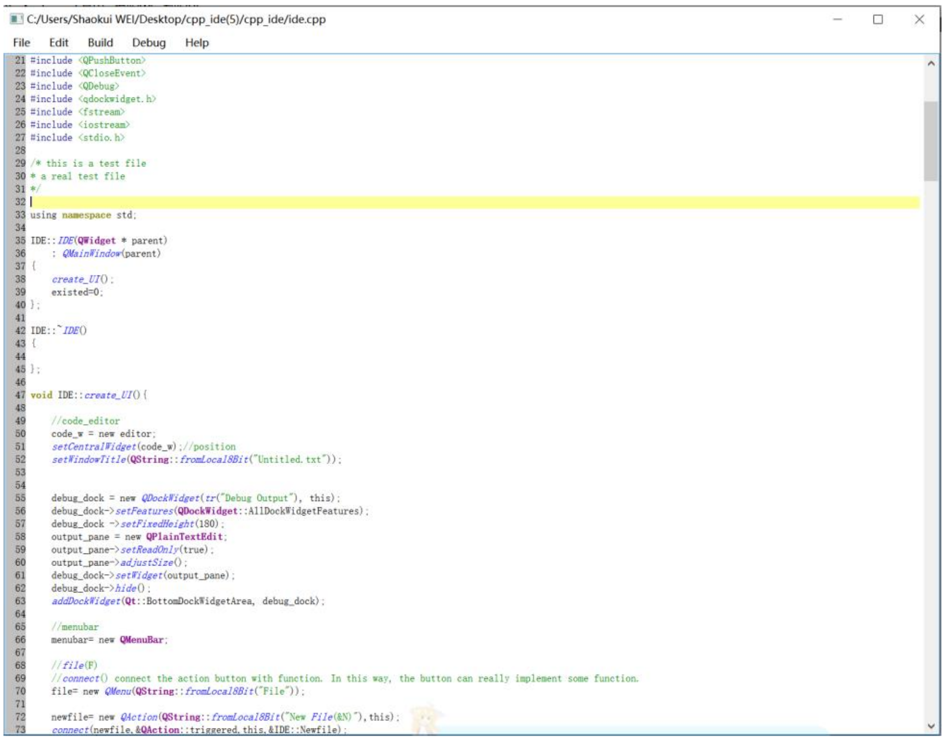


Figure 10

The following functions are implemented in this editor:

- Syntax Highlight:

  Syntax Highlighting in our editor runs by assign colors to reserved string patterns via QSyntaxHighlighter. A series of C++ syntax string pattern is reserved in our program to build the rules of highlighting using regular expressions. Then, QSyntaxHighlighter adopts the rules to realize the syntax highlighting function. An example is shown below:

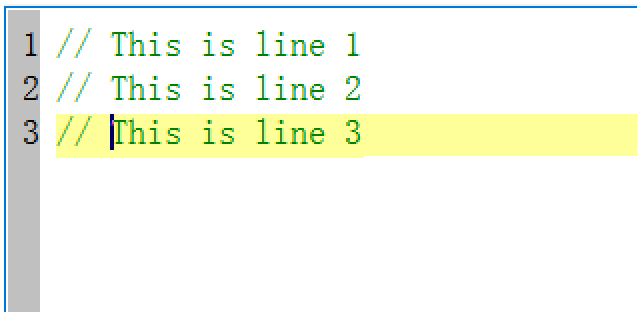


```
#include <QPushButton>
#include <QCloseEvent>
#include <QDebug>
#include <qdockwidget.h>
#include <fstream>
#include <iostream>
#include <stdio.h>

/* this is a test file
* a real test file
*/

using namespace std;

IDE::IDE(QWidget * parent)
    : QMainWindow(parent)
{
    create_UI();
    existed=0;
};

IDE::~IDE()
{

};

void IDE::create_UI(){

    //code_editor
    code_w = new editor;
    setCentralWidget(code_w);//position
    setWindowTitle(QString::fromLocal8Bit("Untitled.txt"));


    debug_dock = new QDockWidget(tr("Debug Output"), this);
    debug_dock->setFeatures(QDockWidget::AllDockWidgetFeatures);
    debug_dock ->setFixedHeight(180);
    output_pane = new QPlainTextEdit;
    output_pane->setReadOnly(true);
    output_pane->adjustSize();
    debug_dock->setWidget(output_pane);
    debug_dock->hide();
    addDockWidget(Qt::BottomDockWidgetArea, debug_dock);

    //menubar
    menubar= new QMenuBar;

    //file(F)
    //connect() connect the action button with function. In this way, the button can really implement some function.
    file= new QMenu(QString::fromLocal8Bit("File"));

    newfile= new QAction(QString::fromLocal8Bit("New File(&N)"),this);
    connect(newfile,&QAction::triggered,this,&IDE::Newfile);
```

Figure 11

- Line Number Display and Current Line Highlight:

  In our editor, we adapted an example from Qt and implemented the line number display and current line highlight. Line numbers are shown in the left border area of the window to help the user locate the lines more quickly. This function is realized by inheriting the LineNumberArea method from the class QWidget. Also, the line where the cursor currently is will be highlighted to help the user locate the cursor faster. An example is shown below:

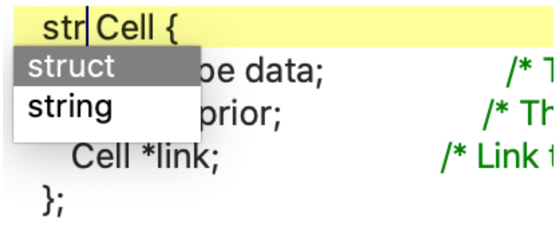


```
// This is line 1
// This is line 2
// This is line 3
```

Figure 12

- Code Completion:

  Another feature about the code editor is the auto completion function, as there is in most popular IDEs. This part is also adapted from an example from Qt. The completer will pop up suggestions for possible words based on the first three characters input by the user and the user's choice of word will be inserted at the cursor position. An example is shown below:

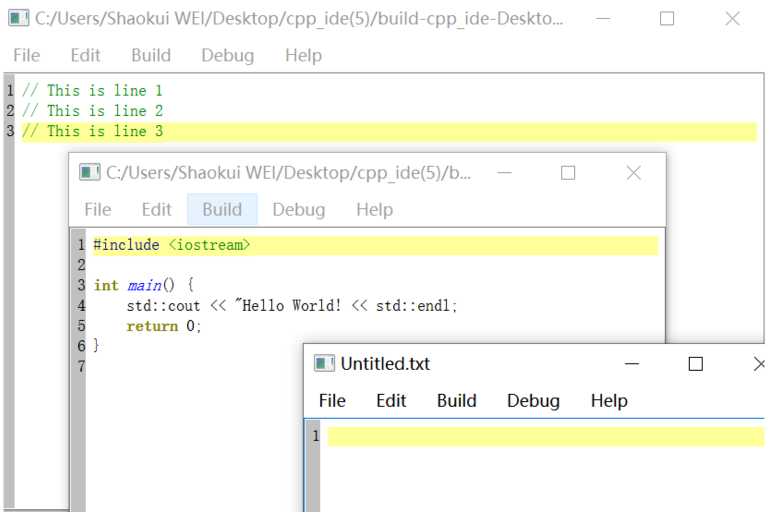


```
str Cell {
struct  pe data;              /* T
string  prior;              /* Th
    Cell *link;           /* Link
};
```

Figure 13

- File Open:

  One important function of our editor is to load existing files from users' computers. First, multiple files can be opened at the same editor and multiple windows will be created to assign each file to a window. Then, although our editor is designed for the text file, it can also load the file in some other format such as .mat, .py, or .cpp.


C:/Users/Shaokui WEI/Desktop/cpp_ide(5)/build-cpp_ide-Deskto...
File
Edit
Build
Debug
Help

```
// This is line 1
// This is line 2
// This is line 3
```

C:/Users/Shaokui WEI/Desktop/cpp_ide(5)/b...
File
Edit
Build
Debug
Help

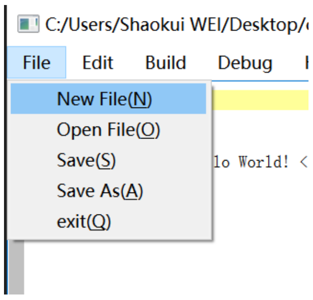
```
#include <iostream>

int main() {
    std::cout << "Hello World! << std::endl;
    return 0;
}
```

Untitled.txt
File
Edit
Build
Debug
Help

Figure 14

- <u>New File</u>

  Besides loading existing files, the user can also create new files via our editor. Each new file will be assigned in a new window.

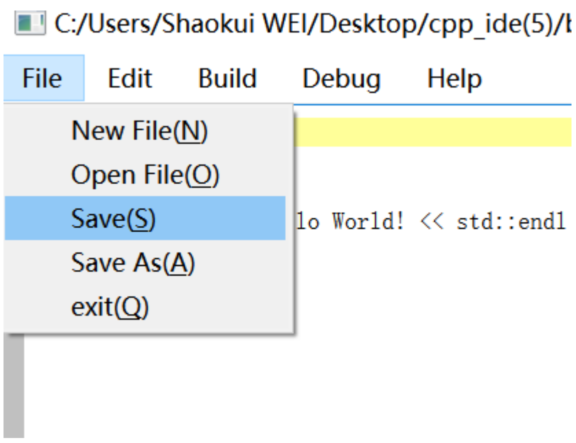


```
lo World! <
```

Figure 15

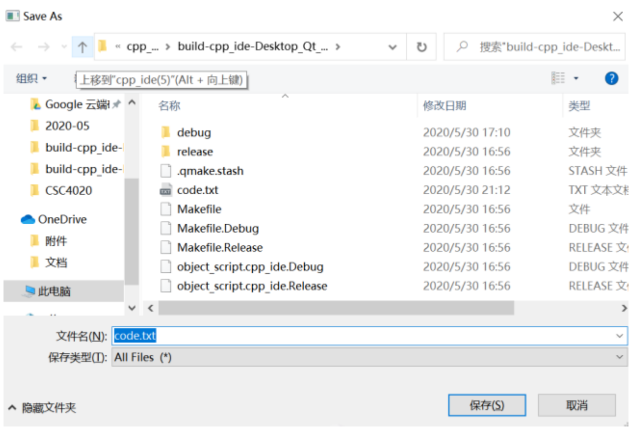
- <u>File Save & Save as</u>

  After editing, users can choose to save the current file either by "Save" or "Save as". The "Save" function will save the current modification to the file for existing an existing file while it will create a dialog window using QFileDialog to get the file path from the user. The "Save as" function will always create a dialog window via QFileDialog to get the new path for the file.



```
lo World! << std::endl
```

Figure 16



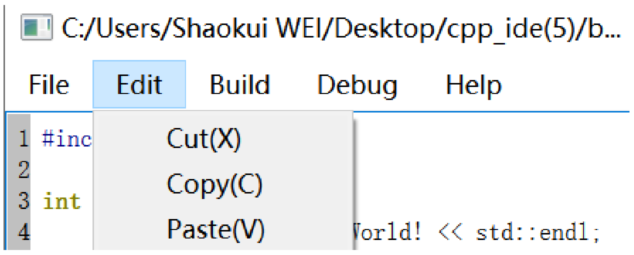
名称	修改日期	类型
debug	2020/5/30 17:10	文件夹
release	2020/5/30 16:56	文件夹
.qmake.stash	2020/5/30 16:56	STASH 文件
code.txt	2020/5/30 21:12	TXT 文本文
Makefile	2020/5/30 16:56	文件
Makefile.Debug	2020/5/30 16:56	DEBUG 文件
Makefile.Release	2020/5/30 16:56	RELEASE 文
object_script.cpp_ide.Debug	2020/5/30 16:56	DEBUG 文件
object_script.cpp_ide.Release	2020/5/30 16:56	RELEASE 文

文件名(N): code.txt
保存类型(T): All Files (*)

Figure 17

- Copy, Cut & Paste

  To facilitate the editing of users, “Copy”, “Cut” and “Paste” are implemented in our editor.

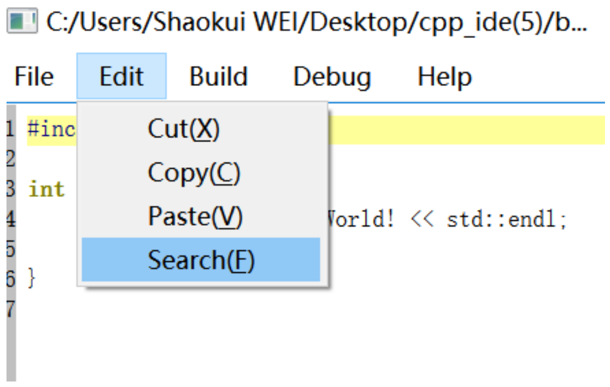


```
#inc

int
orld! << std::endl;
```

Figure 18

- Search

  Like other popular editors, our editor allows users to search for some patterns in the while file. After clicking the search button in the menu, a search window is created as below:



```
#inc
int
                                  orld! << std::endl;
}
```

Figure 19

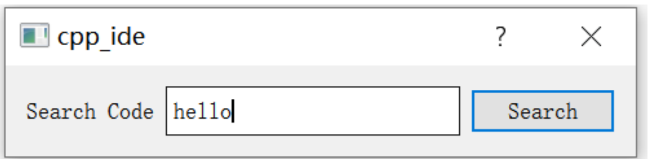
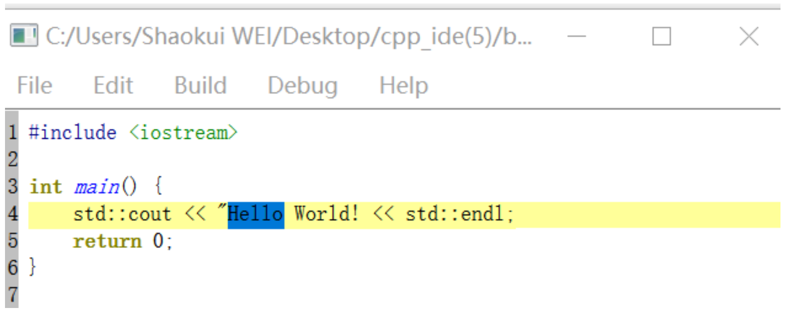


Figure 20

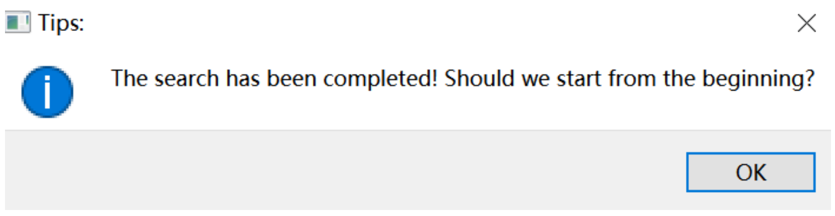
Then, the user can input the content for search and the matched part will be highlighted as below:



```
#include <iostream>

int main() {
    std::cout << "Hello World! << std::endl;
    return 0;
}
```

Figure 21

Also, if there is no matching content in the file, a warning box is shown to the user.



The search has been completed! Should we start from the beginning?

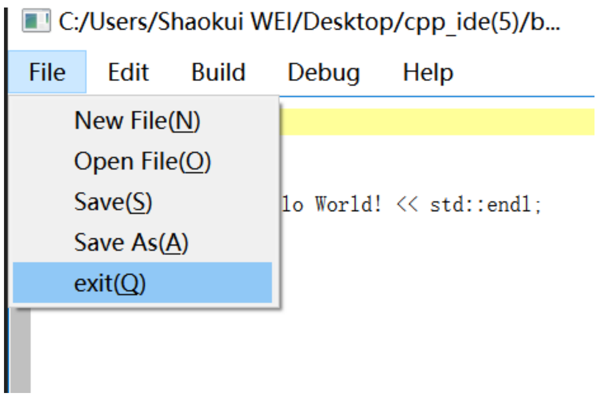
Figure 22

- Run & Debug

  In the menu, "Run" and "Debug" button is designed to run the script and perform debug functions which will be discussed in the later sections.

- Exit:

  When the editing is finished, the user can close the file by clicking the exit button in the menu. Then, our editor will save the file and close the window.



```
lo World! << std::endl;
```

Figure 23

◆ **Preprocessor:**

The process of direct compilation actually requires a strict format of coding. But in real life, seldom people would write their code to meet that standard because it is both inconvenient and visually irregular. So, in order to allow users to code according to their preference, we wrote this preprocessor to process the code they directly wrote and output a code file for direct compilation. In brief, the preprocessor did two things:

- Delete Comments:

  Everything in the comments will be deleted after preprocess. And this program is robust enough to identify and reserve those that look like comments but are actually inside quotation marks.

- Delete and Add Whitespaces:

  The format for direct compilation requires certain whitespaces and rejects some whitespaces. So, the preprocessor also is designed to detect whether whitespace is allowed in the input codes and deal with it accordingly. And it also detects those places where there should be whitespace and decide whether to add one or not.

Moreover, the preprocess can deal with both the usual indent (‘\t\) and C++ coding format indent (‘   ‘).

- **Compiler**

Since they share the functions that execute basic semantic analysis, the compiler and the debugger are developed together though they are implemented separately.

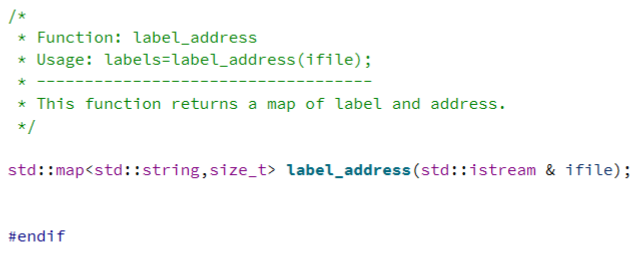
Due to limited time and knowledge on principles of compilers, the compiler only supports the most basic operations in C++. The compiler supports basic mathematical calculations such as addition, subtraction, multiplication, and division. It also supports the “cin” and “cout” operations, and definition and call of functions.

Both compiler and debugger follow this process while dealing with a line of code. Firstly, the program identifies the usage of this line of code. Then according to the type of code, the program executes the following operations. The types considered are the initialization of a variable, initialization of a function, call of a function, the definition of a function, read or print data, and calculation between variables and integers.

The more detailed logic of how the compiler works is as follow: first check whether this line initializes a new variable, if this line initializes a new variable, then check if there are any available registers. If this line is doing a calculation, then check if the involved variables are in registers. If there exist variables that are stored in memory, then check if there are any available registers. If all registers are occupied, then store some data in registers into memory. These steps mainly use the instruction “lw” and “sw”. After the variable is stored into registers, the program checks following operations and output the instructions to implement the operation. If the line of code calls a function, then the program output instructions to save all the progress. Instructions include storing all the data in registers to memory and load parameter data into registers. With all the register cleared and parameter loaded, the program output instruction to jump to the function body to execute the function. After the execution of a function, the return value is temperately stored in register “$v0”, and then moved into target register or memory position. Then the program output instructions to load back data before the execution of functions, restoring the progress.

- **Assembler:**

The basic function of the assembler is to translate MIPS assembly language to machine code. After compiler generating the corresponding MIPS code files of each .cpp and .h file, assembler takes in this collection and generates the machine code files accordingly. A MIPS assembly language file (.asm) basically consists of two sections: .data section and .text section. The assembler handles the .text section only.

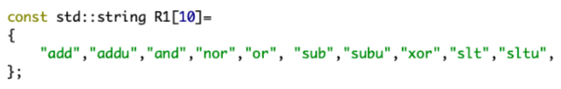


```
/*
 * Function: label_address
 * Usage: labels=label_address(ifile);
 * -----------------------------------
 * This function returns a map of label and address.
 */

std::map<std::string,size_t> label_address(std::istream & ifile);


#endif
```

Figure 24: Symbol table

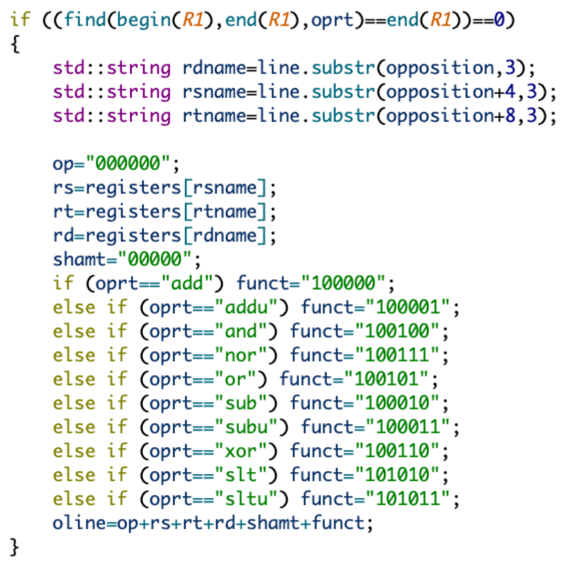
For the .text section, since a label may be used before defined, the address of some labels may be undetermined if the assembler just scans the MIPS file once. Therefore, there has to be a pre-scan. In the first scan, the assembler maps all labels with their address and creates a symbol table using a map named "label_address" (Fig 24). In the second scan, assembler scans and translates the text file to the machine code line by line.

Besides, the instructions are classified according to their format during assembling and the pattern of MIPS code. Taking the example of the R1 group (Fig 25), they all share the pattern of opcode being 0, followed by rs, rt, and rd (Fig 26). This classification greatly improves the efficiency of assembling. And it can support 70 MIPS instructions and syscall.



```
const std::string R1[10]=
{
    "add","addu","and","nor","or", "sub","subu","xor","slt","sltu",
};
```

Figure 25: R1 group

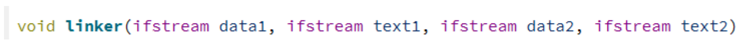


```
if ((find(begin(R1),end(R1),oprt)==end(R1))==0)
{
    std::string rdname=line.substr(opposition,3);
    std::string rsname=line.substr(opposition+4,3);
    std::string rtname=line.substr(opposition+8,3);

    op="000000";
    rs=registers[rsname];
    rt=registers[rtname];
    rd=registers[rdname];
    shamt="00000";
    if (oprt=="add") funct="100000";
    else if (oprt=="addu") funct="100001";
    else if (oprt=="and") funct="100100";
    else if (oprt=="nor") funct="100111";
    else if (oprt=="or") funct="100101";
    else if (oprt=="sub") funct="100010";
    else if (oprt=="subu") funct="100011";
    else if (oprt=="xor") funct="100110";
    else if (oprt=="slt") funct="101010";
    else if (oprt=="sltu") funct="101011";
    oline=op+rs+rt+rd+shamt+funct;
}
```

Figure 26: Assembling R1 group

◆ Linker:

The linker takes two independently assembled machine language programs and "stitches" them together. One keyword of the linker is "relocation". It combines two data sections of the MIPS file and relocation data. The text section is modified base on the relocation information.



```
void linker(ifstream data1, ifstream text1, ifstream data2, ifstream text2)
```

Figure 27

◆ Simulator & Loader:

- Components

  A block of simulated memory (32 bits)

  34 registers (32 general registers and hi & lo)

  The program counter, stack pointer, frame pointer

- Function

  In our program, simulator and loader are combined in the simulator.h file.

  First, the "malloc" function is used to simulate a block of 32-bit memory whose size is 6mb. After that, a PC that points at the lowest position of the text segment is created. Then the loader:

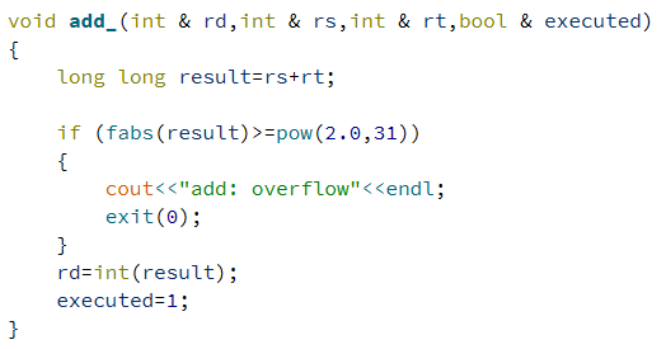
1. It creates an address space large enough for the text and data.

2. Copies the instructions from the executable file into the text segment of simulated memory.

3. Copies the data from the executable file into the data segment of simulated memory.

4. Initializes the registers and sets the stack pointer to the first free location.

After all the data and instructions are loaded into the simulated memory, the simulator begins to execute the code line by line. Each time, PC plus one, then the simulators read the instruction where the PC points to, do corresponding operations base on the instruction. We write a function for each instruction.



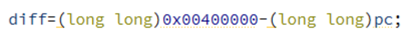
```
void add_(int & rd,int & rs,int & rt,bool & executed)
{
    long long result=rs+rt;

    if (fabs(result)>=pow(2.0,31))
    {
        cout<<"add: overflow"<<endl;
        exit(0);
    }
    rd=int(result);
    executed=1;
}
```

Figure 28: Function for "add" instruction

- Special handling

1. Map 64-bit address with a 32-bit address

We are building a 32-bit MIPS assembler and simulator in this project while our computer's memory is 64 bits. Thus, the simulated 32-bit memory is mapped with 64-bits memory in the real case. The difference between the initial real address and initial simulated address is calculated. When the simulated address is used, the real address can be calculated by simulated address plus difference.



```
diff=(long long)0x00400000-(long long)pc;
```

Figure 29: 32-bit simulated memory and 64-bit memory mapping

2. Void pointer

Since data in MIPS code are of various types, a void pointer that is flexible is used in our simulator. The void pointer is converted to a certain type (e.g. int*, char*, etc) of a pointer using “static_cast” base on the type of data and instructions.

3. Instruction classification

Since there are more than 70 instructions, it is impossible to have a special case for every instruction. However, the characteristics that determine the instruction type are different. For example, the "bgtz" instruction has its opcode being 7 and the 12-to-16 sub-number being 0 (Fig 30) and the “blez” instruction has its opcode being 6 and the 12-to-16 sub-number being 0 as well (Fig 31). So a two-stage classification is applied: first detect the 12-to-16 number, and then detect the opcode. But this kind of process has a drawback at the same time as improving efficiency. An instruction of another type may fall into the wrong first-stage judgment. For example, an “add” instruction with its rt being $zero also has its 12-to-16 number being 0. And after going to the wrong first-stage “if” sentence, it would be hard to get out. So a Boolean variable "executed" is introduced to see if an instruction meets two checks at the same time. Every instruction sets "executed" to true after the execution and the first-stage judgment checks "executed" in the first place (Fig 33).

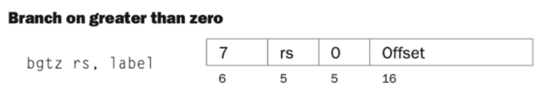


Figure 30: The “bgtz” instruction

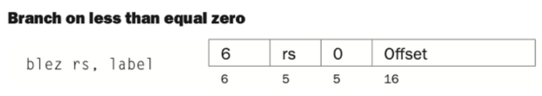


Figure 31: The “blez” instruction

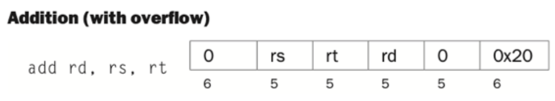
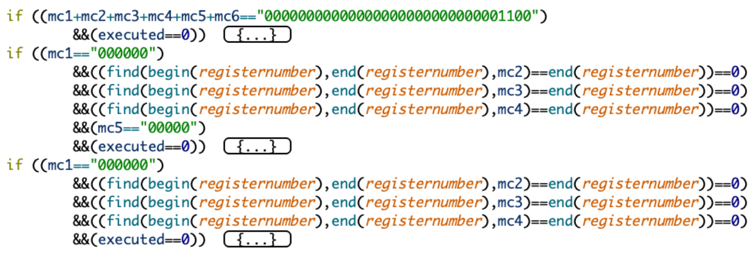


Figure 32: The “add” instruction



```
if ((mc1+mc2+mc3+mc4+mc5+mc6=="00000000000000000000000000001100")
        &&(executed==0))  {...}
if ((mc1=="000000")
        &&((find(begin(registernumber),end(registernumber),mc2)==end(registernumber))==0)
        &&((find(begin(registernumber),end(registernumber),mc3)==end(registernumber))==0)
        &&((find(begin(registernumber),end(registernumber),mc4)==end(registernumber))==0)
        &&(mc5=="00000")
        &&(executed==0))  {...}
if ((mc1=="000000")
        &&((find(begin(registernumber),end(registernumber),mc2)==end(registernumber))==0)
        &&((find(begin(registernumber),end(registernumber),mc3)==end(registernumber))==0)
        &&((find(begin(registernumber),end(registernumber),mc4)==end(registernumber))==0)
        &&(executed==0))  {...}
```

Figure 33: The use of “executed”

◆ **Debugger:**

The debugger is designed relatively simple because of the limited functions the compiler supports. The debugger can check errors in a lack of signs such as parentheses and semicolons, lack of statement, and wrong assignment of value.

Since the compiler relies on some certain signs to identify the different part in a line of code, the first thing the debugger do is to check whether these signs re-used correctly. For regular lines that initialize functions and variables, the debugger checks whether the line lacks a semicolon. For lines that define a function, the debugger checks whether the line lacks parentheses and statements. The debugger can also check whether the value assigned to a variable is correct, by identifying the initialized value type and value.

Contributions:

Name	Main Contributions
Jiayin Liu	Build a GUI and write basic related functions, such as: 1. New, open, save, save as, exit file 2. Copy, cut, paste text 3. Search keywords 4. Highlight text: keywords, classes, functions, references, loading libraries, comments (single line, multiple lines), etc.
Xinyang Tan	1. Connecting Editor with preprocessor in “run” button and with the debugger in “debug” button. Show the syntax check results in a dock widget 2. Line number area and current line highlight in editor

	3. Code completion in editor 4. Optimize program framework
Jiaqi Chen	1. Preprocessor; 2. Several check functions;
Hanzhi Chen	Compiler 1. Basic math calculations 2. Function definition and function call Debugger 1. Check the use of different signs and icons 2. Check the assignment of value
Jianing Liu	1. The fully functional assembler 2. The framework of the simulator and half of the instruction functions
Yudi Yang	Detailed programming of simulator & loader, such as: 1. Realizing some of functions 2. Optimizing and debugging Linker

Reflections:

◆ **Editor:**

I didn't know QT Libraries when I was working on a project. So when designing the GUI, I learned a lot of examples from the QT demo and QT official documents. In the entire QT design process, I learned how to connect the button and the function together, how to design the pop-up window and so on. I think there are a few difficult parts:

1. When designing "save" and "save as" functions, the functional relationship is complicated. I often felt confused during the time I wrote this function. The way I

solved this problem is to get my ideas into shape and continuously optimize my program. Actually, I still wrote many bugs in this part but finally, I managed to fix it.

2. There was also a problem when designing the search bar. If I choose the "FindBackward" method to search the keywords, the program will search backward from the cursor up to the beginning of the article. However, if my cursor is placed at the beginning of the article, the search will end directly, and the search has not achieved the desired effect. Therefore, I combined FindFlag (search forward) and FindBackWard (search backward) to solve this problem together.

3. When designing text highlighting, I learned how to connect two classes to each other, and use "regular expressions" to specify what needs to be highlighted. When designing multi-line comments, I always can't write regular expressions for multi-line comment judgment, which wastes a lot of time here. So I changed the direction to work out this problem. I wrote the regular expressions of "/ *" and "* /" separately, and then find out the content that needs to be annotated in an iterative way.

4. When implementing the code completion feature, the example from Qt only set up the completer using a wordlist text file as the completion resource, which is far from the code completion feature in popular IDEs. So at first I attempted to learn how to make a dynamic resource that the completer can be enriched according to the current code from the user. However, the difficulty is far beyond what I can handle. I only ended up putting all the reserved words of C++ in a file as the completion resource.

There are some additional functions can be implemented in the future:

i. Code Folding: It is useful to collapse sections of code to increase readability in the large files.

ii. More powerful code completion. Currently, the code completion function only applies to the reserved words of C++. In the future, it is necessary to use a dynamic model as the completion resource so that the names of variables and functions defined by the user can be auto-completed.

iii. Rename refactoring: Sometimes, users need to rename a symbol such as a function name or variable name. With a rename refactoring function, users can rename all instances in a short time.

iv. Errors and Warnings: It’s of great help to users if the errors and warnings are highlighted as users edit their codes with squiggles.

◆ **Preprocessor:**

When writing the preprocessor, at first, I was only trying to extend the function deleteComment() to detect and reserve all the content within quotation marks. But after discussion, along with the teammate who is responsible for the compilation, we set up a coding standard for compilation. But we would like to give our users more freedom to code according to their habits, and even allow room for some non-grammar mistakes. Therefore, the function deleteComment() was largely extended and named as Preprocess.

The first difficulty came up when I was facing the choice between get() and getline(). For different lines of codes, there might be different procedures to be applied. So, it is vital to first get some information about the line we are dealing with before we actually do anything with it. The initial method to deal with the code I used was get(). But for judging the nature of a specific line, I need to use getline() to get the line as a string. But that would conflict with my method of putting things into the output file and even the deleting of the comment. After searching on the internet, I learned about methods tellg() and seekg() under class ifstream, then I decided to use both get() and getline(). For every line of the input codes, the program read a line with getline(), and uses other check functions (some wrote by me and some wrote by another teammate) to determine the nature of the line. And we use seekg() to get back to the beginning of the line with the coordination pregiven by tellg(). Then we can get back on our original get() method to take in and output the codes.

Then the second difficulty came up when I basically finished writing the codes. Before I ran my code, I was actually quite confident about it. But the first output was completely not understandable. So, I dug in the If branches and started the painful journey of debugging. I wrote debug update for every run of my program, and fix thing one bit by another. After at least 20 runs, I am confident about the rest of my code but one part. And that's when the third and the most terrible difficulty in this program.

The third difficulty was a very strange problem of which I couldn't find the cause. The output of my program, at that time, misses the first few digits in every line of code. And it even followed a special pattern. For the first line, everything is fine. But since the second line of input code until the second last line, some front digits are missed. And from the second last line to the second line, the number of digits missed increases from 1 by 1. After testing with multiple other independent programs, I found that the program misses those digits exactly in tellg()-getline()-seekg() process. And the tellg() seemed to return a false location of the cursor. I cudgeled my brain that night until 4 a.m. to try to find the cause. I search online but no one has ever reported such a problem. I did all changes that might help, but nothing changes that bug. The next day, after I woke up, I gave it to my teammate and ask him to run the program for me. And it just magically worked out. After a few communications and some small changes, it is finally ready to work for our project.

This group project experience has taught me a lot. First of all, the internet is a good thing. The problems we had are very likely somebody else's problems too. So, it is important to make good use of the searching engine and learn from others' experiences. Second, debugging is actually the process where we truly get to fully understand our program. When writing codes, many ideas would constantly come up to your mind. So, chances are we will change our original thoughts and code another way. Sometimes this would cause unknown problems. And debugging will make us face those problems, and get the full picture of our program. Last but not least, the most important thing I learned from this project is teamwork. When I am at the most despairing situation, the responses and help of my teammates are really the most treasurable things. Fighting alone and fighting with companions are totally different. And such experience certainly tells me the importance of teamwork.

- **Compiler:**

The main difficulty in implementing this compiler is the allocation of register and the semantic analysis. The semantic analysis is done by finding certain words or signs at certain places. Since in this program logic branches and other even more complicated functions are not supported, the work is simplified. As mentioned before, the code that the program could deal with can be separated into several types. The program treats different types of codes differently. The allocation of a register is done with the help of several containers. These container records the status of different registers and the position where the variable is stored. With this information, the program can use registers correctly.

Another difficulty I met with was the assembling language. Use instruction to implement a project is very different from using a language like Python or C++. Therefore, how to use a program to translate code in C++ into MIPS assembling language is a challenge. The solution I choose is to treat code line by line. For the simplest codes, this is an efficient and relatively simple way to implement the translation. However, this method is not flexible enough and could make the work even tougher when a program becomes more complex. Therefore, I need more study into the principles of compilers for further research and learning.

This project helps me learn more about how programming language and compiler co-work, and finally make a program run a computer. Despite the fact that this is a programming course, it helps me obtain a deeper understanding of how software and hardware work together step by step.

- **Assembler:** The main difficulty lays in the comprehension of the instruction format, summarizing the pattern of instructions and grouping the instructions accordingly, totally understanding how assembler works and handling the address involved instructions. It is a big work to go through 70 instructions and look into each of them. It took a very long time to do the research and summarize it by myself. I also got very confused when I finished the very first version that can be compiled. I went through the output machine code line by line and bit by bit to find what went wrong. After fixing the problems caused by a typo, I found all the addresses wrong. It was because there have to be the mapping between labels and address in advance. However, it cost me a long time to figure this out and make the assembler functional.
- **Linker & Simulator & Loader:** The simulator is the key for the functional IDE and it is one of the most tricky parts. The coding was too massive and one little mistake can shut down the whole simulation, leading to no output of the IDE. The first step is to handle with the MIPS code file and process the file to a data session and a text session with all the unwanted blank spaces, tabs, symbols, and empty lines removed, so that all the lines are in one pattern. It took me long to handle all the possible input to reduce the work of generating MIPS code files in a particular format. The second step is to load the data and the instructions into memory. This is very complex because different variable types have to deal with, and they have different sizes and working principles. A lot of pointer-related stuff was used. Among all, the string is the most difficult to deal with because the characters are stored 4-by-4. The third step is to judge the instruction type, which is the reverse process of assembling. The grouping is alike but trickier because there have to be

two or three or even more conditions fulfilled simultaneously to have the instruction name determined, so that two different instructions may satisfy one initial condition at the same time. More precise and reliable grouping is needed. The fourth step is writing the syscall. Syscall is the core to have the output. Saving and loading words are the most difficult parts. The fifth step is to finish the rest of the instructions. To avoid using too many pointers, we chose to use calling by reference on the registers instead of declaring int pointers for registers. This actually simplifies the work. The most difficult part is using the pointer of a pointer to make the PC jump. It took me very long to fully understand how does this work. After all of these are done, the debugging of the simulator is even more tiring because there are too many details, it is a very big part. I printed out the address and the variables after every instruction executed to see what went wrong. There were many unexpected problems. The one that worth talking about the most is that I wrote the functions of converting binary strings to integer and converting an integer to binary string very early, and I ensured they were all functional. But the last problem was that the PC cannot jump to the expecting address even if all the other things were right. The true reason is that when I convert a negative number to positive, I added 2 to the power of digit, but I used this number as an int, which will encounter overflows while dealing with the address of the long long type. I tried to debug very often to avoid debugging a lot at one time, but it is kind of difficult for this situation because the test file can only run well with all the instructions available. Looking back, maybe I could write more simple test files by myself. And it turns out that I spent too much time on the functions that the compiler cannot support. I should have communicated with them earlier to have our time better allocated.