PIC 10A
Miscellaneous

Ernest Ryu
UCLA Mathematics
Hardware

To build a computer, you need to the following components:

- CPU
- RAM
- HDD/SSD
- Motherboard
- Case
- Power supply
- Graphics card and monitor
- Keyboard and mouse
Hardware

- The motherboard electronically holds the components together.
- The case physically holds the components together.
- The power supply converts potentially unstable AC power from the wall into stable DC power.
- The graphics card and monitor handles the visual output. How exactly is beyond the scope of this class.
- You know exactly what the keyboard and mouse do.
Hardware

- The CPU (central processing unit) is the brain of the computer. It performs the elementary operations (e.g. copy something from here to there, add these two ints, execute this if this bool is true) that form a complex program.
- RAM (random access memory) is the primary storage of the computer. You use RAM to store information you need until the end of a program.
- HDD (hard disk drive) or SSD (solid state drive) are the secondary storage of the computer. You use secondary storage as permanent storage.
Basic computer architecture

CPU + RAM + HDD/SSD is the minimal way to think of a computer.
Primary vs. secondary storage

- RAM is fast while HDD and SSD are slower. Roughly, the read/write speeds are 10GB/s vs. 100MB/s and 500MB/s. Furthermore, RAM (random access memory) has fast random access.
- RAM is not permanent; it’s erased when the power is shut off. HDD and SSD are semi-permanent storage.
- Variables in C++ (and any programming language) are not meant to be permanent. They are stored in memory.
- A program primarily works on RAM. You save to the HDD/SSD only when you come by something worth saving.
How does RAM work?

The CPU sends 32 (or 64) bits memory address. This allows the CPU to read or write the 1 byte stored in that location. (By the way, this is why a 32bit CPU can only use 4 GB of RAM.)
For HDD and SSD, each address corresponds to a sector. A sector contains 512B or 4kB. Even if you only need a few bytes from a sector, you still have to spend the time accessing the other bytes. (This is why HDD and SSD have poor random access speeds.)
In programming, a number written in base 2 is said to be written in *binary*.

\[
27 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\
= 11011 \text{ (base 2)}
\]

A number written in base 8 is said to be written in *octal*.

\[
27 = 3 \times 8^1 + 3 \times 8^0 \\
= 33 \text{ (base 8)}
\]

(Octal isn’t used much any more.)

A number written in base 10 is said to be written in *decimal*.

\[
27 = 2 \times 10^1 + 7 \times 10^0 \\
= 27 \text{ (base 10)}
\]
Base 16

Binary uses 2 single digit numbers, octal uses 8, and decimal uses 10. A *hexadecimal* number, also called a *hex* or base 16 number, uses 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

\[ 27 = 1 \times 16^1 + 11 \times 16^0 \]
\[ = 1B \text{ (base 16)} \]

For example, the largest 4 digit hexadecimal number is

\[ \text{FFFF (base 16)} = 15 \times 16^3 + 15 \times 16^2 + 15 \times 16^1 + 15 \times 16^0 \]
\[ = 65535. \]
Base 16 in programming

Each hexadecimal digit corresponds to 4 binary digits. Two hexadecimal digits correspond to a byte (=8 bits).

\[ F3 \text{ (base 16)} = 1111,0011 \text{ (base 2)} \]

Hexadecimal numbers are common in programming, because the conversion to binary is convenient. For example, it is clear that 3AAB,F043 can be represented with 32 bits. That 984,346,691 (base 10) can be represented with 32 bits is not clear.
Compiling in 64-bit

By default, VS 2015 compiles code into 32-bit applications, which pointers are 32 bits long, even on a 64-bit machine. You can choose to compile in 64 bits.

- Right click your project and select Properties.
- Click Configuration Manager on top right corner.
- In the dropdown menu below Active Solution Platform, choose x64.

```cpp
int main() {
    const double d = 1;
    cout << typeid(&d).name() << endl;
    cout << sizeof(double*) << endl;
    cout << &d << endl;
}
```
Specification

A *specification* or *spec* is a requirement your product should or does meet.

“This charger will provide an output of 5V if provided with an input of 100–240V from the wall.”
Real-world engineering (including programming) is done with specs.

- The requirements of the homework problems can be considered specs.
- Specs usually say: “If the input is X, your program will do Y.”
- Rigorous specs clearly define what your product does and under what condition your product functions properly.
- We say we go off spec when the specified conditions are not met. The product may or may not function reasonably off spec. (Who knows how the Apple charger functions when it’s given an input of 80V.)
- Programming languages are specifications themselves. “If you write code according to rule X, your code will do Y.”
History of C++

The C++ standard/spec has been growing over time.

72 The programming language C was created.

78 Bjarne Stroustrup started the project “C with classes”, an extension of C.

83 Stroustrup renamed his project to C++ (incremented C).

98 C++ standardized by the International Organization for Standardization (ISO). This version nicknamed C++98.

03 New version, nicknamed C++03. (Mostly bug fixes.)

11 Major update, nicknamed C++11. Many new features. Until its release, this version was called C++0x (as it was thought to be finished by 09).


17 Newest version. Was called C++1z.
Bjarne Stroustrup

- Born 1950, Denmark
- PhD 1979, Cambridge
- 1979–2002, Bell Labs
Dennis Ritchie (at Bell labs) started working on C, as an improvement upon the programming language B.

C was standardized in 1989–1990, and is nicknamed C89 or C90.


Some quality of life updates. Nicknamed C11. Was called C1X.
Relationship of C and C++

C is (almost) a subset of C++.

- You can compile a C program with a C++ compiler.
- C is very minimal. C++ inherits the syntax from C.
- Sometimes, there is a “C-style” way of doing something, inherited from C, and an improved way, made possible by the C++ syntax.
- Programming in C is like cooking a complex dish with one (good) knife and one (good) pan. It’s difficult and awkward.
- C++ has so many features to accommodate various project needs and programming styles. Almost no one uses all features.
- C++ also inherits standard libraries from C. For example, the C library `math.h` is called the `cmath` library in C++.
- By the end of this class, you will have learned 90% of C.
- A good C++ programmer must also be fluent in C because many C++ programmers (unfortunately) program in C-style. After this course, you can spend a few hours to learn C and become fluent.
C vs. C++

Always use C++ when you can.
"This is somewhat controversial: [C] is obsolete." — Stroustrup

A C compiler is much easier to make than a C++ compiler since C has so fewer features. (Providing a knife and a pan is easier than providing a kitchen full of tools.) Because of this, sometimes only C is available, and you have no choice.
To code in C++, you need
  ▶ a text editor
  ▶ and a compiler.
These are the bare minimum, although other tools are helpful.
C++ without an IDE

For example, I use Emacs (a text editor) and g++ (a C++ compiler) to code on my Linux machine. Note the simplicity. No “solutions”. No “projects”.

![Emacs and g++ on Linux](image)
C++ without an IDE

For example, I use Emacs (a text editor) and g++ (a C++ compiler) to code on my Linux machine. Note the simplicity. No “solutions”. No “projects”. 
C++ without an IDE

For example, I use Emacs (a text editor) and g++ (a C++ compiler) to code on my Linux machine. Note the simplicity. No “solutions”. No “projects”.

![Image of terminal output showing Emacs and g++ commands]

eryu@kona:~/cpp$ emacs test.cpp
eryu@kona:~/cpp$ ls
test.cpp
eryu@kona:~/cpp$ g++ test.cpp
eryu@kona:~/cpp$ ls
a.out test.cpp
eryu@kona:~/cpp$ ./a.out
Hello world.
eryu@kona:~/cpp$
Microsoft Visual Studio is an *integrated development environment* (IDE).

- An IDE is an all-in-one program for coding; it’s a single program that contains a text editor, a compiler, a debugger, and many other useful tools.
- Xcode is an IDE provided to MacOS users.
- You don’t need an IDE to write code, and some argue beginners shouldn’t use IDEs for pedagogical reasons.
Programmers perpetually debate whether IDEs are good or evil.

- An IDE can cost money. They are bulky to install and run.
- Some think IDEs are overkill. Some take (snobbish) pride in their minimalistic approach to programming.
- IDEs make complex program structures more manageable.
- Because of autocompletion, IDE users prefer descriptive but long variable names. Non-IDE users prefer short variable names.
- Not using an IDE encourages simple and concise approaches.
- A non-IDE programmer must procure the individual tools like the text editor, compiler, debugger, etc. (I use 5 separate tools to code a small project on Linux.)
- IDEs are necessary for large, complex projects.

IDEs are useful tools. Learning how to use VS is part of this course.
A *bug* is an error in a program. *Debugging* is the effort to find and eliminate bugs. The *debugger* is a particular tool used in debugging. VS provides a great debugger.

Some rely on *cout* to check the state of a program when debugging, and take pride in their bare-handed approach. I find debuggers too useful to eschew.
VS debugger: breakpoints

You set a *breakpoint* in your code with Debug → Toggle Breakpoint, with the hotkey F9, or by clicking on the left side of your code.

Once you set a breakpoint, you can run your program in debug mode with Debug → Start Debugging or with the hotkey F5. Your code will run until you hit a breakpoint.

Once at a breakpoint, you can continue to the next breakpoint with Debug → Continue or with the hotkey F5.
You can *step* through your code line-by-line with Debug → Step Over or with the hotkey F10. Most of the time, you’ll want to go to a breakpoint and then start stepping.

Step Into, bound to F11, is similar to Step Over, but it will enter function calls. I’ll demonstrate Step Into once we cover functions.
While in debug mode you can monitor local variables with
Debug → Windows → Autos
and
Debug → Windows → Locals.

Autos window shows the recent variables (including ones on the heap).
Locals window show all local variables in scope.

To monitor variables on the heap, first open the Diagnostic Tools windows with Debug → Windows → Show Diagnostic Tools. Then Memory Usage → Take Snapshot → View Heap.
Pseudocode is an informal high-level description of a program. For example, the following pseudocode illustrates how to compute $n!$

```
prod = 1
for i = 1,2,...,n
    prod = prod * i
endfor
```

This “code” is not written in any particular programming language. Rather, it gets across the main point of how to write the for loop to compute $n!$. 
Edge case

An edge/corner case is a problem or situation that occurs only at an extreme operating parameter.

- In the palindrome problem of hw3, an input of 0 (which is a palindrome) is an edge case.
- In the hex to binary problem of hw4, an input of 0x0 is an edge case.
- For example, the quadratic formula

\[ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

doesn’t handle the edge case of \( a = 0 \).

- Some edge cases are obvious. Some are totally unexpected.

You must handle edge cases well to write reliable programs. Reliability matters in the real world.

(You must handle edge cases well to get perfect marks in this class.)
C++ is a compiled language: C++ code must be compiled into an executable before it can be run. Roughly speaking, the compilation comprises 4 stages:

1. Preprocessing
2. Compilation (somewhat confusing)
3. Assembling
4. Linking

Preprocessing is processing done before the actual compilation. Details of the last 3 stages are beyond the scope of this class.

Languages like Python are interpreted languages, which execute directly without compilation. Interpreted languages tend to be more convenient but slower.
Preprocessor directives (things like `#include` or `#define`) start with the hash character `#`.

- A preprocessor directive ends at the end of the line, not a semicolon. (So only one preprocessor directive per line.)
- The preprocessor handles the preprocessor directives first and passes on the result to the compiler.
- Preprocessor directives are agnostic to the C++ syntax.
There are 2 ways to use `#include`

```c
#include <cmath>
#include "my_header.h"
```

Directive: “copy code from file X and paste it here.”

The convention is to use `#include <X>` to include standard libraries and `#include "X"` to include files you have written.
#define

The preprocessor directive

```
#define identifier replacement
```

defines the identifier as a *macro* and replaces any instance of it with replacement.

As mentioned, `#define` is indifferent to the C++ syntax and will mechanically replace the macro with its replacement. `#define` is C-style, and using it in C++ is considered bad programming practice. Usually, there is a better C++-style way.
For example, we can use `#define` for a *C-style constant*.

```
#define PI 3.1415926535897932
int main() {
    cout << "Input a radius\n";
    double r; cin >> r;
    cout << "The area of the circle is"
    << (r*r*PI) << endl;
}
```

In C++, it’s better to use `const`.
For example, we can use `#define` to switch between single or double precision in computation.

```cpp
// #define float_type float
#define float_type double
int main() {
    cout << "Input a radius \n";
    const float_type PI = 3.1415926535897932;
    float_type r; cin >> r;
    float_type output = r*r*PI;
    cout << "The area of the circle is" << output << endl;
}
```

In C++, there is a better way (`typedef`).
You now understand 2 parts of the hello world program: "Hello world" and #include <iostream>.

```cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Hello world" << endl;
}
```

You will understand all of it by the end of this term.
In programming, you will often encounter a tradeoff between the hours you spend programming vs. the program’s speed.

- You don’t always have to take the hard road. (Speed doesn’t always matter.)
- Your time is valuable. There’s no shame in choosing the easy way out.
- You are selling features, not code. It’s important that your product, not your code, is good.
Should you use C++ or, say, Python?

- C++ is difficult to use but the product is fast. Most time-critical programs (e.g. video games) are written in C++.
- You are much more “productive” with Python. I.e., you can write the same (but much slower) application much faster.

“[C++] is notoriously difficult to learn and use.” — MIT Technology Review
“There are just two kinds of languages: the ones everybody complains about and the ones nobody uses.” — Stroustrup
Compile-time vs. run-time

The workflow of coding: write code → compile code → run program. *Compile-time* is before and at the moment of compilation. *Run-time* is when you run the program.

- Compile-time errors cause compilation to fail.
- Run-time errors manifest themselves when running the program.
- For example, division by 0 is usually a run-time error.
- In C++, the length of a local array must be specified at compile-time. The length of a dynamic array can be provided at run-time.

This is a simple but useful distinction.
Good errors

Fail *early*, fail *loudly*. Such errors are easier to catch.

▶ Compile-time errors are early (compile-time) and loud (program doesn’t compile).
▶ Good programming habits (e.g. using `const`) make errors more likely to be early and loud.
▶ Memory leaks are terrible because they are late and silent.
assert(condition) is a debugging tool that allows you to catch run-time errors. If condition is not true, the program will hault (which is definitely loud).

```cpp
#include <iostream>
#include <cassert>
using namespace std;

int main() {
    cout << "Input a positive number" << endl;
    int i; cin >> i;
    assert(i>0);
}

(You can tell from the name that cassert is inherited from C.)
**cassert**

assert is a great debugging tool.

- assert conditions that you think must be true. You might be wrong.
- Use assert profusely.
- You may not want asserts to halt a final product. There’s a way to automatically remove them when compiling.
- There are more elegant ways to handle errors (e.g. with “exceptions”) but we won’t cover them in this class.
Undefined behavior

I say a code is illegal (not a technical term) if there’s no rule that makes it legal (and doesn’t compile) or if the behavior is *undefined*.

A familiar example with undefined behavior is

```c
int i;
cout << i << endl;
```

The behavior may differ depending on what computer you use, the state of your computer, and what compiler you use.

Errors caused by undefined behavior are often silent and therefore difficult to catch.
Some compilers offer *extensions*, which are additional features not mandated by the C++ standard. For example,

```cpp
#include <iostream>
using namespace std;

int main() {
    int n; cin >> n;
    double d[n];
}
```

is illegal under the C++ standard and won’t compile in VS 2015.

However, g++ (a Linux and MacOS compiler) allows this as an extension. You can run into portability issues when you use extensions.
Compilers allow the user to specify certain options such as:

- which C++ version to compile in
- whether to compile in 32 or 64 bits (which we saw how to do in VS)
- whether use nonstandard extensions and
- whether or not to warn you of weird stuff.

We won’t talk much about this. Just be aware this issue exists.
Although this program is completely legal

```cpp
#include <iostream>
using namespace std;

int main() {
    int a;
    cout << "hello" << endl;
}
```

by default, VS will warn you of the unused variable a. The wise heed advice. Don’t turn off warnings.
Compiler settings: nonstandard extensions

For example, g++ on MacOS has the -pedantic option to check if you’re using nonstandard extensions.

VS 2015 warns you of nonstandard extensions by default.
Compiler settings: nonstandard extensions

For example, `g++` on MacOS has the `-pedantic` option to check if you're using nonstandard extensions.

```cpp
#include <iostream>
using namespace std;

int main() {
    int n; cin >> n;
    double d[n];
}
```

VS 2015 warns you of nonstandard extensions by default.
Compiler settings: nonstandard extensions

For example, g++ on MacOS has the -pedantic option to check if you’re using nonstandard extensions.

VS 2015 warns you of nonstandard extensions by default.
Portability

Code that strictly adheres to the C++ standard is portable. I.e., such code can be compiled on any system and will behave the same.

This is only 99.9% true, and the programmer can always make mistakes. In practice, the only way to be sure your code behaves the same on another system is to try it.